VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Analýza Bezpečnosti JavaScript knihoven a rámců

Security Analysis of JavaScript Libraries and Frameworks

Bc. Tomáš Zvolánek

Diplomová práce

Vedoucí práce: Ing. Jan Plucar, Ph.D.

Ostrava, 2021

Abstrakt

Diplomová práce se zabývá bezpečnostní analýzou balíčků a knihoven psaných v programovacím jazyce JavaScript. Účelem práce je popsat aktuální bezpečnostní problémy a zranitelnosti JavaScriptového ekosystému. Práce zároveň rozebírá současnou metodiku řešení daných problému a nedokonalosti těchto řešení. Součástí práce jsou také dva experimenty. První z nich se zaobírá testováním existujícího vzorku škodlivého skriptu. Druhý experiment obsahuje implementaci sedmi útoků a zranitelností a jejich následnou analýzu třemi volně dostupnými nástroji.

Klíčová slova

JavaScript; skript; npm; malware; analýza; bezpečnost

Abstract

This diploma thesis describes the process of security analysis of JavaScript packages and libraries. The thesis aims to discuss current security issues and vulnerabilities in the JavaScript ecosystem while also outlining state of the art solutions to these problems as well as the flaws of the solutions themselves. Two experiments have been conducted as a part of the thesis. The first experiment focuses on testing an existing malware sample. The second experiment is based on the implementation of seven attacks and vulnerabilities and their subsequent analysis with three open-source tools.

Keywords

JavaScript; script; npm; malware; analysis; security

Poděkování

Na tomto místě bych rád poděkoval všem, kteří mi s prací pomohli, především Ing. Janu Plucarovi, Ph.D. za hodnotné rady a vedení této práce.

Obsah

Seznam použitých zkratek a symbolů

XSS	–	Cross Site Scripting
npm	–	Node Package Manager
DOM	–	Document Object Model
HTTP	–	Hyper Text Transfer Protocol
ES6	–	ECMAScript 2015
ES5	–	ECMAScript 2009
CVE	–	Common Vulnerabilities and Exposures
URI	–	Uniform Resource Identifier
URL	–	Uniform Resouce Locator
ASCII	–	American Standard Code for Information Interchange
API	–	Application Programming Interface
CI	–	Continuous Integration
CD	–	Continuous Deployment
SPA	–	Single Page Application
HTTPS	–	Hyper Text Transfer Protocol Secure
MD5	–	Message Digest Algorithm 5
SVG	–	Scalable Vector Graphics
MB	–	Mega Byte
SHA	–	Secure Hashing Algorithm
OS	–	Operační Systém

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

V roce 2020 se v prostředí internetu nachází stovky milionů webových stránek a aplikací, které poskytují obsah a služby miliardám uživatelů. Značná část těchto aplikací přitom ke svému fungování využívá právě programovací jazyk JavaScript, který je možné aplikovat jak na straně klienta, tak na straně serveru. JavaScript je jedním z nejpopulárnějších programovacích jazyků současnosti a je zároveň součástí značného počtu rámců, knihoven a navzájem závislých balíčků, které jsou k tvorbě webových aplikací a služeb využívány.

Knihovny a balíčky jsou často open-source a jsou tak vývojářům dostupné bez jakékoli finanční investice. Součástí JavaScriptového ekosystému jsou také nástroje pro správu balíčků, díky kterým je instalace konkrétního balíčku do projektu otázkou jedno příkazu. Všechny tyto vlastnosti urychlují a usnadňují vývoj JavaScriptových aplikací.

Zmíněné vlastnosti však přinášejí i řadu bezpečnostních rizik. Balíčky a knihovny jsou často pod správou několika málo vývojářů, kteří ne vždy dokáží včas odstranit veškeré zranitelnosti nacházející se v daných repozitářích. Balíček navíc může obvykle publikovat kdokoli a není nijak zajištěno, aby kód balíčku prošel bezpečnostním auditem. Dalším problémem jsou škodlivé balíčky, které jejich autoři vytvořili právě se záměrem zavést do aplikace zranitelnost.

Cílem práce je popsat problémy JavaScriptového ekosystému, způsoby analýzy JavaScriptového kódu a zároveň probrat překážky, se kterými se analýza JavaScriptových komponent potýká. První kapitola se věnuje úvodu do probírané problematiky a její závažnosti. Druhá kapitola poskytuje základní informace o vlastnostech jazyka JavaScript, na které práce navazuje v pozdějších kapitolách. Následujících několik kapitol se zaobírá metodikou analýzy kódu a detekcí škodlivých skriptů. Poslední kapitoly teoretické části práce popisují způsoby, jakými se zranitelný kód šíří a nakonec také zranitelnosti a útoky samotné. Praktická část práce obsahuje dva experimenty, které se zabývají testováním a analýzou škodlivých skriptů.

Kapitola 2

State of the Art

V dnešní době nabízí JavaScriptový ekosystém vývojářům nepřeberné množství knihoven a balíčků, které jsou zároveň z velké části open-source. V těchto knihovnách a balíčcích, které jsou na sobě často závislé, mohou být zranitelnosti, kterých může útočník využít. Balíčky jsou denně distribuovány v řádu stovek milionů stažení denně, v případě neobjevené zranitelnosti tak tvoří znatelné riziko. Jen z balíčkového registru *node package manager* (dále jen *npm*) bylo v srpnu 2017 stahováno v průměru přes 375 milionů balíčků denně. V dokumentu The State of Open Source Security společnosti Snyk z roku 2017 jeho autoři uvádí, že při testování 430 000 webových stránek se ukázalo, že 77% z nich používá nějakou knihovnu, ve které existuje známá zranitelnost. Zároveň uvádí, že medián času, po kterém je takováto zranitelnost od zavedení do knihovny odhalena, je 2,5 roku. Jako medián času potřebného na opravu dané zranitelnosti uvádí 16 dní. [1]

Dokument The State of Open Source Security společnosti Snyk z roku 2020 poukazuje na růst JavaScriptového ekosystému, kdy v roce 2019 přibylo v registru *npm* přes 300 000 nových balíčků. Přes 80% zranitelností nalezených v *npm* ekosystému bylo důsledkem nepřímé závislosti, kde uživatelé nepoužívali balíček se zranitelností přímo, ale jako závislost jiné knihovny, která byla v projektu zahrnuta. [2]

Nejčastějším útokem na webu v posledních letech stále zůstává tzv. cross-site scripting. Cross-site scripting (dále jen „XSS“) je typ útoku, při kterém útočník vloží do webové stránky škodlivý skript, který je následně spuštěn na straně klienta, v prohlížeči oběti. Tímto způsobem může útočník měnit funkcionalitu stránky, případně krást a odesílat data oběti. Podle společnosti Trustwave byl XSS v roce 2017 předmětem zhruba 40% webových útoků. Základní prevence proti tomuto typu útoku je ošetření uživatelských vstupů. [2, 3]

Skutečnost, že je nějaký typ zranitelnosti četnější než jiný však nemusí nutně znamenat to, že také páchá větší škody. Takovým případem je podle společnosti Snyk například zranitelnost zvaná prototype pollution. Tato zranitelnost měla dle jejich výzkumu větší dopad na projekty, ve kterých se vyskytla, než právě zranitelnost vůči XSS útoku a byla obsažena například v populárních knihovnách jakými jsou jQuery nebo Lodash. [2]

Riziko představují také tzv. škodlivé balíčky (malicious packages). Jsou to takové balíčky, které byly buď přímo navrženy pro zavedení zranitelnosti do aplikace, která je použije, nebo balíčky, které byly škodlivým kódem pouze kontaminovány. [2]

Co se obrany a předcházení těmto útokům týče, existuje několik mechanismů, které mohou být v tomto směru poměrně efektivní. Jedním z nich je statická analýza kódu, kdy je zdrojový kód podroben analýze ještě před tím, než je spuštěn. Statická analýza má za cíl objevit v kódu chyby různého charakteru, včetně těch bezpečnostních a umožnit tak jejich opravu v raném stádiu vývoje. Pro tento účel bylo vyvinuto mnoho nástrojů se zaměřením na různé programovací a skriptovací jazyky. Integrovaná vývojová prostředí dnes již rovněž nabízí podobné nástroje a vývojář tak může dostávat zpětnou vazbu v průběhu samotného psaní kódu. Nástroje navíc proces automatizují a usnadňují tak provádění analýzy nad velkým objemem zdrojových souborů. Tímto způsobem je vhodné hledat například útoky typu přetečení bufferu nebo SQL injekce, naproti tomu problémy s autentizací nebo nevhodné využití kryptografie je takto obtížné detekovat. [4, 5]

Statická analýza kódu však naráží na dynamické vlastnosti JavaScriptu, jakými jsou například načítání kódů za běhu programu nebo dynamické typovaní proměnných. Dynamická analýza, při které je skript spuštěn a až následně analyzován, může být pro analýzu JavaScriptového kódu lepší volbou. [6]

Dalším důležitým mechanismem jsou nástroje pro hledání závadných závislostí, které procházejí strom závislostí dané aplikace a následně porovnávají verze balíčků a knihoven se záznamy v databázích zranitelností. Za tímto účelem vzniklo několik open-source, ale i komerčních nástrojů. Například v ekosystému *npm* je v prostředí příkazového řádku možné využít příkazu *npm audit*, který projde závislosti daného balíčku a vrátí zprávu o zjištěných zranitelnostech a případných aktualizacích. Slabými místy jsou v tomto případě jednotlivé databáze zranitelností, které ne vždy obsahují aktuální informace a může se stát, že se v nich nově nalezené zranitelnosti objeví například až za několik týdnů či měsíců po jejich objevení. [7, 8, 9]

Kapitola 3

JavaScript

Práce se zabývá striktně knihovnami a balíčky napsanými v jazyce JavaScript. Z tohoto důvodu je vhodné pro porozumění pozdějších kapitol uvést zde některé jeho technické detaily, které útočníci využívají k napadení aplikací.

3.1 Historie a současnost

JavaScript je objektově orientovaný, interpretovaný jazyk s objekty založenými na prototypech. Tento programovací jazyk je dále charakteristický dynamickou typovou kontrolou a tzv. *first-class functions*. JavaScript vznikl v roce 1995 ve snaze rozšířit funkcionalitu webových stránek v prohlížeči Netscape Navigator, dnes je však využíván v naprosté většině moderních webových aplikací a prohlížečů, jakožto implementace standardu ECMAScript (dále jen ES). V případě dnešních prohlížečů se už těžko hledá takový, který by v sobě neměl zabudovaný JavaScriptový interpretr. Díky popularitě projektu *Node.js* se v posledních letech stal JavaScript velmi rozšířeným i mimo prostředí webových prohlížečů a stal se tak jedním z nejpoužívanějších programovacích jazyků současnosti. [10, 11, 12]

Od jeho vzniku navíc velmi výrazným způsobem narostl počet open-source projektů třetích stran psaných právě v JavaScriptu. Do těchto projektů se řadí jednoduché balíčky sestávající z několika řádků kódu spravované jednotlivci, ale také rozsáhlé knihovny pro tvorbu webových aplikací jakými jsou například *Angular* od společnosti Google nebo *React* vyvíjený společností Facebook. [13]

Jedním ze středových bodů tohoto open-source ekosystému je balíčkový registr *npm*, který poskytuje online databázi obsahující v únoru 2021 již přes jeden a půl milionu JavaScriptových modulů. Týdenní počet stažení modulů se pohybuje v řádu desítek miliard. V únoru roku 2019 obsahoval registr npm zhruba 800 000 modulů, jejich počet se tak za poslední dva roky téměř zdvojnásobil. [7]

3.2 Vykonávání JavaScript kódu ve webových prohlížečích

Jedním z důvodů, proč má JavaScript tak velký potenciál stát se škodlivým je způsob, jakým webové prohlížeče vykonávají JavaScriptový kód. Webový prohlížeč totiž po navštívení stránky automaticky předpokládá, že je skript důvěryhodný a bez jakékoli kontroly ho spustí, včetně případných skriptů třetích stran. Útočník tak může získat částečnou kontrolu nad webovým prohlížečem a získat skrze něj přístup k výpočetním zdrojům a datům uživatele. [14]

Existují však restrikce, které omezují působnost JavaScriptu ve webovém prohlížeči. Webový prohlížeč má velmi omezenou možnost přistupovat k souborovému systému uživatele. Prohlížeč může pracovat s jednotlivými soubory a adresáři pouze pokud je uživatel sám vybere. Operace, které je možné nad těmito soubory a adresáři provést závisí na implementaci jednotlivých prohlížečů. Existuje několik dalších mechanismů, které jsou v moderních prohlížečích implementovány pro zvýšení bezpečnosti. [15]

3.2.1 Same-origin policy

Většina moderních prohlížečů má v základním nastavení aktivní tzv. *same-origin policy*, což je mechanismus, který omezuje interakci skriptů jedné domény s jinou doménou. Same origin policy se vztahuje také na síťový port a protokol. Tento mechanismus zabraňuje například tzv. *cross-site request forgery* útoku, kde se útočník snaží přimět oběť poslat HTTP požadavek, kterým provede změny ve webové aplikaci, kde je oběť autentizována. [16]

3.2.2 Cross-Origin Resource Sharing

Cross-origin resource sharing je mechanismus založený na použití speciální HTTP hlavičky, díky které může server rozhodnout, jakým doménám a rozhraním bude poskytovat přístup k jeho zdrojům. [17]

3.3 JavaScript na straně serveru

Využití JavaScriptu není omezeno pouze na klientskou část aplikace. Běhové prostředí Node.js je příkladem populární technologie, která umožňuje běh JavaScriptu na straně serveru. Pro správu modulů třetích stran využívá Node.js ve výchozím nastavení již zmíněný npm. Tento registr open-source balíčků však žádným způsobem neručí za jejich obsah, proto se může jednoduše stát zdrojem zranitelností.

Jedním z útoků je injekce JavaScriptového kódu do běhového prostředí aplikace, ten je pak v kontextu serveru vykonán. V tomto případě má skript oproti jeho vykonávání v prohlížeči rozdílné možnosti. Útočník může potenciálně přistupovat k souborovému systému serveru, spouštět binární

soubory, navázat vzdálené připojení k serveru a v případě Node.js dokonce přistoupit ke zdrojům operačního systému. [18]

3.4 Separace privilegií

Na rozdíl od jiných programovacích jazyků nemá JavaScript žádný mechanismus, kterým by oddělil práva balíčků třetí strany od práv samotné aplikace. Jinými slovy, pokud je v rámci aplikace spuštěn balíček třetí strany, pak balíček disponuje stejnými právy, jako zbytek aplikace, a to i v případě, že je využíván jako nepřímá závislost. Tato vlastnost je problematická zejména mimo prostředí webových prohlížečů, u těch je totiž působnost JavaScriptového kódu izolována na samotný prohlížeč. [7]

3.5 Strict mode

Strict mode je omezená podmnožina JavaScriptu, která byla vytvořena jako součást ES5 (ECMAScript 2009) za účelem eliminace chyb starších verzí, které není kvůli zpětné kompatibilitě možné odstranit. Strict mode tak zajišťuje lepší kontrolu chyb a zvýšenou bezpečnost kódu. Kód, který je spouštěn v kontextu strict mode se označuje termínem *strict code*. Pro deklaraci takového kódu se používá direktiva *use strict*, kterou je nutné uvést buď na začátku skriptu, nebo na začátku těla funkce. Pokud je kód součástí ES6 (ECMAScript 2015) modulu nebo třídy, je automaticky považován za strict code. [10]

3.6 Prototypy

Většina objektů v jazyce JavaScript disponuje vlastností zvanou *prototype*, která odkazuje na další objekt. Tato vlastnost má několik využití. Pokud chceme například přistoupit k vlastnosti libovolného objektu, která však neexistuje, JavaScript se pokusí onu vlastnost nalézt právě u objektu, na který odkazuje vlastnost prorotype. Tento krok se opakuje, dokud není nalezena buď daná vlastnost, nebo konec prototype řetězce. Na konci řetězce je pak většinou *Object.prototype*, což je objekt, který nabízí různé pomocné funkce, jako například *toString()*. Tímto způsobem je v JavaScriptu realizována dědičnost. Zároveň lze mechanismus zneužít, protože pokud se útočníkovi podaří dynamicky modifikovat prototype vlastnost objektu, modifikuje tak i všechny další objekty v prostředí aplikace, které jsou součástí jeho prototype řetězce. Tento útok se nazývá *prototype pollution* a je detailněji rozebrán v kapitole 8.1. [19]

Pro úplnost je potřeba zmínit také vlastnost *___proto___* a funkci *setPrototypeOf()*. Jak ___proto___, tak *setPrototypeOf()* se využívají k modifikaci prototypu objektu. Funkce *setPrototypeOf()* přišla se standardem ES6 a podle JavaScriptových konvencí by se měla používat právě místo zastaralé vlastnosti ___proto___.

3.7 Node Package Manager

Balíčkový open-source registr *npm* je v dnešní době nedílnou součástí JavaScriptového ekosystému. Npm je největší softwarový registr na světě a skládá se ze tří hlavních komponent:

- Webová stránka, která slouží k vyhledávání a správě balíčků, uživatelských účtů a organizací.
- Příkazový řádek slouží pro interakci uživatele s *npm*. Uživatel může skrz příkazový řádek instalovat a konfigurovat závislosti aplikace, případně provádět jejich bezpečností audit.
- Nedílnou součástí *npm* je samotná databáze JavaScriptových balíčků a jejich metadat. Databáze aktuálně obsahuje přes jeden a půl milionu balíčků a počet týdenních stažení se pohybuje v řádu desítek miliard. [20]

3.7.1 Publikování balíčků

Npm umožňuje jakémukoli registrovanému uživateli sdílet kód se zbytkem platformy. Pro publikování balíčku stačí mít vytvořený uživatelský účet, následně vytvořit soubor *package.json*, což je soubor s metadaty balíčku. Nakonec je potřeba v adresáři se souborem package.json použít pomocí *npm* příkazového řádku *npm publish*.

3.7.2 Instalace balíčků

Pro stažení a nainstalování jakéhokoli veřejného balíčku (*npm* umožňuje placeným uživatelům svůj balíček sdílet pouze určitým organizacím) pak stačí jediný příkaz: *npm install*. Příkaz nainstaluje daný balíček a s ním všechny jeho závislosti.

Kapitola 4

Metody analýzy JavaScriptového kódu

Analýza počítačového programu je proces, při kterém se automatizovaně zkoumá chování programu. Proces se dělí do dvou hlavních kategorií, kterými jsou statická a dynamická analýza. Cílem kapitoly je popsat principy těchto způsobů analýzy a jejich nedostatky zejména při analýze JavaScriptového kódu.

4.1 Statická analýza

Metoda statické analýzy se používá pro analýzu programu bez toho, aniž by bylo nutné program spustit. Statická analýza kódu je často součástí procesu revize kódu (anglicky *code review*) v implementační fázi softwarového vývoje. Pro statickou analýzu jsou běžně využívány nástroje, které se snaží ve statickém zdrojovém kódu (před spuštěním kódu) nalézt chyby a zranitelnosti pomocí technik jakými jsou například tzv. *taint analysis* nebo *data flow analysis*. [21]

4.1.1 Taint Analysis

Předmětem tohoto typu analýzy je vyhledat proměnné, které obsahují data vložená uživatelem. Využívání takových proměnných v kódu činí aplikaci zranitelnou vůči útokům jako XSS (viz kapitola 7.1) nebo SQL injection. Výpis 4.1 představuje triviální příklad kódu, který by měl být během analýzy zachycen. [21]

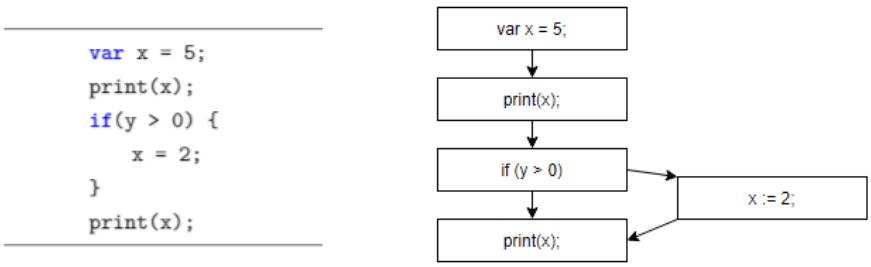
```
<input type="text" id="user-input"/>
<button onClick="vulnerableFunction();"> Kliknutí spustí uživatelský vstup metodě
    eval() </button>
<script>
    function vulnerableFunction() {
        let input = document.getElementById("user-input");
        eval(input.value);
```

```
    }
</script>
```

Listing 4.1: Příklad XSS zranitelnosti XSS kvůli neoštřenému uživatelskému vstupu.

4.1.2 Data Flow Analysis

V tomto případě jde o analýzu toku dat za běhu programu. Informace o tomto toku jsou však opět získávána na základě statického kódu. Průběh analýzy lze reprezentovat pomocí tzv. *control flow graph*. Control flow graph (viz obrázek 4.1) je graf, jehož vrcholy reprezentují sekvenci instrukcí a hrany grafu reprezentují skoky mezi nimi. [21]



Obrázek 4.1: Pseudokód a jeho abstrakce pomocí control flow grafu.

4.1.3 Lexikální Analýza

Lexikální analýza převádí zdrojový kód na sekvenci tzv. *tokenů* ve snaze kód abstrahovat, kdy zároveň odstraní komentáře a prázdné znaky. Tokeny následně usnadňují syntaktickou kontrolu kódu. Příklad tokenů pro JavaScriptový kód je na obrázku 4.2. [21]

```
//JavaScriptový kód
var testVariable = "Lexikální analýza";

//Převod na tokeny
TOKEN_KEYWORD TOKEN_IDENTIFIER TOKEN_OPERATOR TOKEN_STRING
```

Listing 4.2: Příklad tokenů v JavaScriptovém kódu.

Statická analýza je škálovatelná a velmi efektivní při hledání určitých typů zranitelností, jakými jsou například přetečení bufferu, nebo SQL injekce. Zároveň přibývá nástrojů, které je možné integrovat do vývojových prostředí a analýzu tak automatizovat. [21]

Tento typ analýzy kódu však má i své nevýhody a není vhodný například pro identifikaci problémů s autentizací nebo kryptografií. Dalším problémem může být vysoká míra jak falešně pozitivních, tak i falešně negativních výsledků. [21]

JavaScript navíc disponuje celou řadou dynamických vlastností, které statickou analýzu dále komplikují:

4.1.4 Dynamické typování

Na rozdíl od staticky typovaných jazyků jako Java nebo C++ může v JavaScriptu proměnná za běhu programu nabývat hodnot různých typů. JavaScript sám o sobě neposkytuje žádnou typovou kontrolu. Kvůli některým operacím navíc dochází k implicitním typovým změnám, kde hodnota mění svůj typ tak, aby bylo možné operaci provést. [22]

4.1.5 Dynamické načítání kódu

Webové aplikace běžně načítají skripty z externích serverů například pomocí HTML značky *<script/>*, nebo pomocí XMLHttp požadavku, který je načítán asynchronně. Takový kód není snadné analyzovat staticky, protože jeho načtení probíhá až po spuštění kódu aplikace. [22, 6]

4.1.6 Dynamické prvky

JavaScript obsahuje několik metod, které mohou za běhu programu spouštět kód, který byl načten ve formě řetězce znaků, rovněž za běhu programu. Příkladem mohou být metody *eval* a *setInterval*. Existují nástroje, jakými jsou například *Unevalizer* nebo *Evalorizer*, které transformují eval na jiné konstrukty JavaScriptu a usnadňují tak statickou analýzu. [22]

4.1.7 Dynamické objekty

JavaScript umožňuje dynamicky přidávat a odebírat také vlastnosti objektů, kvůli tomu je statickou analýzou obtížné určit, v jakém bodě běhu programu jsou tyto vlastnosti dostupné. Dynamicky je možné generovat dokonce i identifikátory vlastností. [6]

4.1.8 Document Object Model a Eventy

Jelikož je JavaScript stále převážně využíván v prohlížečích, musí být statická analýza uzpůsobena také metodám DOM API a eventům. Taková analýza je však extrémně složitá, protože eventy jsou většinou spouštěny uživatelskou akcí a jejich běh je asynchronní. [22]

4.2 Dynamická analýza

Dynamická analýza zkoumá kód po jeho spuštění. V případě JavaScriptu to znamená, že odpadá hned několik problémů, na které narážela analýza statická. Tento typ analýzy pozoruje chování spuštěného kódu, který již obsahuje konkrétní typy proměnných, funkce a vlastnosti objektů. Dynamická analýza si také dokáže poradit s dynamickým načítáním kódu, běh skriptu se totiž začne analyzovat až v okamžiku, kdy je skript načten. Oproti statické analýze může být nevýhodou fakt, že je analyzován pouze určitý počet konkrétních instancí spuštění programu. [6]

Stejně jako statická analýza, i dynamická analýza naráží na několik problémů v podobě vlastností jazyka JavaScript:

4.2.1 No Crash Philosophy

Článek [6] úvádí jako jeden z problémů dynamické analýzy JavaScriptu tzv. *No Crash Philosophy*. JavaScript se totiž před uživatelem snaží skrýt některé chyby, například pád programu. Jako další příklad uvádí násobení řetězce znaků s datovou strukturou pole, kdy JavaScript nezobrazí žádnou chybovou zprávu i přes to, že se jedná o nesmyslnou operaci. Kvůli této absenci zobrazování chyb může být pro dynamickou analýzu obtížné rozlišit zamýšlené a chybné chování programu. [6]

4.2.2 Nedeterminismus

Aplikace založené na JavaScriptu, zejména webové aplikace, často obsahují velké množství uživatelských vstupů a interakcí se servery. Jak už bylo zmíněno výše, dynamická analýza se soustředí pouze na limitovanou množinu spuštění programu, proto nemusí pokrýt veškeré chování způsobené kombinací uživatelských akcí. K nedeterminismu JavaScriptových aplikací navíc přispívá také nemalý počet asynchronních zpráv a eventů u kterých není zřejmé, v jakém pořadí budou vykonány. [6]

Kapitola 5

Antivirová detekce škodlivých souborů a skriptů

Různé antiviry používají pro detekci škodlivých souborů a skriptů různé metody, existuje však několik základních funkcí, které jsou obsaženy ve většině antivirových produktů. Předmětem kapitoly jsou dvě z nich: Detekce založená na signaturách a heuristická analýza. Kapitola obsahuje popis těchto metod včetně způsobů, jakými se malware snaží před metodami skrýt.

5.1 Detekce nebezpečného kódu pomocí signatur

Signatury jsou typicky hashe (například SHA) nebo byte-streamy, které obsahují informace, pomocí kterých je možné identifikovat nebezpečný soubor nebo skript. Tento způsob detekce využívá pro jeho jednoduchost a rychlost většina antivirových produktů, které si signatury uchovávají ve svých databázích. [23]

Antiviry používají pro generování signatur různé algoritmy. Následuje výčet a popis vybraných algoritmů pro tvorbu signatur:

5.1.1 Byte-streams

Jednou z nejjednodušších podob signatury je byte-stream, který je specifický pro škodlivý soubor. Tento způsob je rychlý a jeho implementace je nenáročná. Nevýhoda přístupu spočívá v tom, že jako malware bude označen jakýkoli soubor s tímto byte-streamem, nezávisle na tom, jestli je skutečně škodlivý. [23]

5.1.2 Cyklický redundantní součet

Cyklický redundantní součet (anglicky Cyclic Redundancy Check) je algoritmus, který je běžně používán pro detekci změny nebo ztráty dat. Algoritmu jsou jako vstup předány data, ze kterých

algoritmus vygeneruje hash ve formě kontrolního součtu (anglicky checksum). Tento hash má pouze několik bytů, v závislosti na verzi použitého CRC algoritmu. Jako vstup může být algoritmu předán celý program, nebo jen jeho část. Výstupní hashe se následně mohou porovnat s již dříve uloženými hashi vzorku malware. [23]

Výhoda algoritmu je jeho rychlost. Nevýhodou je velký počet falešně pozitivních výsledků. CRC nebyl stvořen pro účely detekce škodlivých programů, nýbrž jako detekce poškozených a neúplných dat. Algoritmus totiž nemusí nutně generovat unikátní výstup pro rozdílné vstupy. I z tohoto důvodu antiviry ustoupily od používání CRC ve prospěch robustnějších *kryptografických funkcí*. [23]

5.1.3 Kryptografické funkce

Kryptografické funkce na rozdíl od CRC algoritmu generují pro každý unikátní vstup unikátní výstup a tím snižují počet falešně pozitivních výsledků. Tato vlastnost však může být považována i za nevýhodu, protože útočník může změnou jednoho bitu učinit program nedetekovatelným, protože výsledný hash bude naprosto odlišný od toho původního. Výpočty v kryptografických funkcích mohou být zároveň oproti algoritmu jako CRC náročnější, a tím pádem pomalejší. [23]

5.1.4 Fuzzy Hashing

Fuzzy hashing svými vlastnostmi spadá na pomezí výše zmíněných postupů.

- Pokud se nepatrně změní vstupní hodnota algoritmu, měla by se nepatrně změnit také výstupní hodnota a to jen pro korespondující bloky dat.
- Je jednoduché identifikovat vztah mezi vstupními daty a vygenerovaným hashem.
- Počet kolizí mezi hashi je závislý na konkrétním případu. Výsoký počet kolizí může být například vhodný pro detekci spamu, naopak pro detekci malwaru je vhodnější nízký počet kolizí.

Tento typ algoritmu by měl při správné konfiguraci a implementaci zajišťovat menší množství falešně pozitivních výsledků. Zároveň je obtížnější signatury tohoto typu obejít, jelikož útočník musí program výrazně pozměnit hned na několika místech, protože malá změna výsledek algoritmu nezmění. [23]

5.2 Heuristická analýza

Detekce na bázi signatur je nevhodná pro rychle se vyvíjející a polymorfní viry, které neustále mění své vlastnosti a chování, ve snaze uniknout detekci. Pro tento úkol je vhodnější aplikovat nástroje heuristické analýzy, které hledají podezřelé chování mezi zatím neklasifikovanými a modifikovanými vzorky malware. Nástroje pro heuristickou analýzu můžeme rozdělit na statické a dynamické, jejich kombinací pak vzniká nástroj hybridní. [23]

5.2.1 Statické heuristické nástroje

Statické heuristické nástroje mohou být implementovány různými způsoby. Jedním ze způsobů je implementace založená na principu strojového učení, kdy se využívá algoritmů jakými jsou například Bayesovská síť, nebo genetických algoritmů. Tyto algoritmy jsou vhodné pro nalezení společných charakteristik pro velké rodiny virů. Nejsou však nejlepší volbou pro desktopové antivirové produkty, protože generují vysoký počet falešně pozitivních výsledků a spotřebovávají velké množství zdrojů. Z těchto důvodů jsou vhodné spíše pro laboratorní podmínky. Pro desktopové produkty jsou vhodnější tzv. *expert systems*. [23]

Expert system je heuristický nástroj, který využívá algoritmů, které imitují rozhodovací proces při analýze člověkem. Člověk může například rozpoznat, jestli Windows PE (portable executable) program jeví známky malware i bez nutnosti pozorovat jeho chování při spuštění a může si vystačit s analýzou struktury souboru nebo s analýzou výsledku tzv. *disassembly* procesu. Člověk by hledal známky obfuskace, anti-debugging techniky nebo techniky používané pro zmatení uživatele, jako například změnu základní ikony pro PE soubor. V případě nalezení těchto artefaktů by pravděpodobně shledal soubor podezřelým a dále by jej analyzoval více do hloubky. Expert system je nástroj, který takové rozhodování implementuje. [23]

5.2.2 Dynamické heuristické nástroje

Při dynamické heuristické analýze je program spuštěn ve virtualizovaném prostředí a následně analyzován. Výhodou dynamické analýzy je odolnost vůči polymorfním a metamorfním virům. Nevýhodou je delší čas potřebný pro analýzu oproti analýze statické. Další nevýhodou je samotné virtualizované prostředí, protože pokud malware dokáže rozpoznat, že byl v takovém prostředí spuštěn, může pozměnit své chování a tím analýzu zkomplikovat. [23]

Dynamické nástroje často pro monitorování běhu programu používají tzv. *háky* (z anglického *hooks*). Hák je místo v kódu nebo v řetězci volání zpráv, kam je možné vložit vlastní kód, který změní chování systému. V případě OS Windows je možné použít háky například pro API volání jako *CreateFile* nebo *CreateProcess*. Místo těchto funkcí se nejprve spustí antivirový kód, který v závislosti na konkrétní implementaci běh programu povolí, zablokuje a nebo nahlásí. [23]

Existuje několik druhů háků, princip jejich fungování však bývá podobný:

1. Antivirová knihovna je vložena do procesů, které je třeba monitorovat. Typicky je knihovna vložena do všech procesů a dochází tak k monitorování celého systému.
2. Antivir si určí, které API metody chce monitorovat.
3. První instrukce dané API metody jsou nahrazeny instrukcemi, které zapříčiní skok do kódu antiviru.
4. Po doběhnutí antivirového háku je kontrola předána zpět původnímu řetězci volání API.

Kapitola 6

Nástroje pro analýzu a detekci

Cílem kapitoly je stručně představit několik nástrojů jak pro statickou, tak pro dynamickou analýzu souborů. Všechny zde popsané nástroje jsou součástí experimentů v kapitole 9, proto bude součástí popisu také zdůvodnění, proč byl nástroj vybrán.

6.1 ESLint

ESLint je jedním z nejpopulárnějších open-source linterů pro JavaScript. Linter je typ nástroje pro statickou analýzu kódu, který v kódu hledá prvky, které nesouhlasí s předem určenými pravidly. ESLint disponuje přes 200 pravidly, která se dělí do několika sémantických kategorií. Vývojář má plnou kontrolu nad tím, která pravidla bude jeho projekt používat. ESLint navíc vývojáři dovoluje napsat a přidat vlastní pravidla. [24, 5]

Důvodem pro volbu tohoto nástroje je především jeho popularita. V registru npm zaznamenal ESLint již stovky milionů stažení a má aktivní komunitu na platformě GitHub. Druhým důvodem je flexibilita nástroje, tedy možnost upravovat kontrolovaná pravidla podle potřeby.

6.2 Virus Total

VirusTotal je bezplatná služba pro bezpečnostní analýzu souborů a URL. Tato služba umožňuje uživatelům zadat URL, případně nahrát soubor z jejich počítače a následně nahrané položky analyzuje pomocí 70 antivirových služeb. Výsledky analýzy jsou pak dále sdíleny nejen s uživatelem, ale také se spolupracujícími antivirovými společnostmi. VirusTotal poskytuje webové i desktopové rozhraní, rozšíření prohlížeče a HTTP API pro přístup pomocí jakéhokoli programovacího jazyka. [25]

VirusTotal byl pro účely této práce vybrán, protože představuje agregaci velkého vzorku antivirových služeb a tím značným způsobem usnadňuje analýzu testovaných skriptů.

6.3 Cuckoo Sandbox

Cuckoo Sanbox je open-srouce automatizovaný systém pro analýzu malware. Tento nástroj umožňuje nahrát a analyzovat velké spektrum podezřelých souborů a webových stránek v různých virtualizovaných prostředích. Cuckoo nabízí statickou i dynamickou analýzu chování souborů a poskytuje jak desktopové, tak webové rozhraní. Nástroj u analyzovaného souboru vypíše výčet podezřelého chování a přiřadí souboru skóre, kde skóre 0 znamená neškodný soubor, naopak skóre 10 značí vysoce podezřelý soubor. [26]

Cuckoo Sandbox byl zvolen zejména pro jeho schopnost spustit nahrané skripty ve virtualizovaném prostředí a provést dynamickou analýzu souboru.

Kapitola 7

Distribuce zranitelného a škodlivého kódu

Velmi podstatnou částí životního cyklu nebezpečného a zranitelného kódu je jeho distribuce, tedy jakým způsobem může do webové aplikace či k uživateli proniknout. Cílem kapitoly je popsat metody distribuce a obranných mechanismů, které mohou šíření skriptů zabránit.

7.1 Cross-site scripting

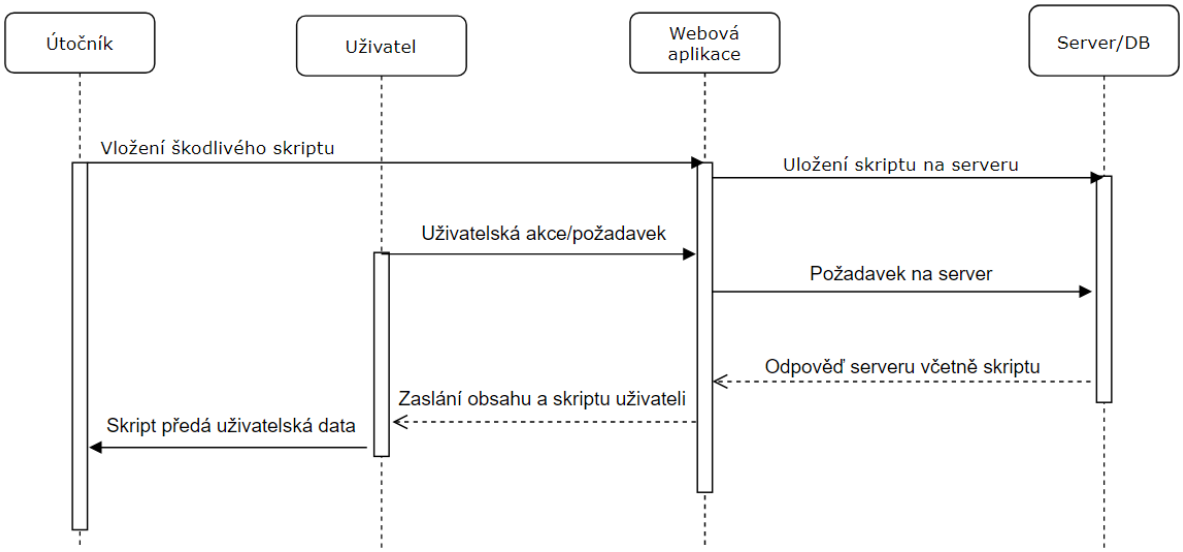
Útoky typu Cross-site scripting jsou v současnosti stále nejčetnějšími útoky v prostředí webových aplikací. V projektech monitorovaných společností Snyk tvořil v roce 2019 XSS 18% všech nahlášených útoků. Společnost Trustwave uvádí, že XSS v roce 2017 představoval dokonce 40% útoků na webu. [2, 3]

Základním principem XSS je vložení škodlivého skriptu do webové stránky. Když pak uživatel stránku navštíví, je skript v prohlížeči spuštěn. Zdrojem skriptu je důvěryhodná stránka a samotný prohlížeč nedokáže škodlivost skriptu rozpoznat. Útočník tak může skrze skript získat uživatelská práva, přístup k citlivým informacím, případně je schopen měnit funkcionalitu stránky. XSS útoky se dělí do 3 kategorií: [27, 28]

- *Stored XSS* útok (viz obrázek 7.1) se vyznačuje tím, že je skript útočníka trvale uložen na serveru webové aplikace, například v databázi. Když uživatel následně požádá server o uložená data, server odešle uživateli skript, který se v jeho prohlížeči vykoná. [27, 28]

- *Reflected XSS* útok nastává tehdy, když jsou uživatelské vstupy aplikací okamžitě zpracovány a zaslány zpět uživateli bez toho, aby byly náležitě kódovány. [27, 28]

- *DOM based XSS* je méně známou variantou XSS útoku. V tomto případě se nemění obsah HTTP odpovědi serveru, ale modifikuje se pouze prostředí DOM. [29]



Obrázek 7.1: Sekvenční diagram Stored-XSS útoku.

7.1.1 Obrana

Základním principem obrany proti XSS útoku je omezit počet míst, kam může uživatel vložit data a zároveň ošetřit ty uživatelské vstupy, jejichž existence je nezbytná. Open Web Application Sercurity Project (dále jen OWASP) dokonce doporučuje model, kde se až na určité výjimky zakazuje jakékoliv vkládání nedůvěryhodných dat do HTML dokumentu aplikace.

Webové aplikace se dnes však často neobejdou bez interakce s uživatelem a uživatelskými vstupy. Proto je nezbytné tyto vstupy ošetřit. Jakým způsobem vstupy ošetřit je závislé na kontextu, do kterého jsou data ze vstupů vkládána. Jiná pravidla například platí pro znaky a řetězce vložené přímo mezi dvě HTML značky, HTML atributy, nebo pro JavaScriptové skripty. Pro každý z těchto kontextů je nutné nahrazovat různé množiny znaků tak, aby je aplikace interpretovala jako text a ne jako příkaz. Některé kontexty je však velmi obtížné, nebo dokonce nemožné korektně ošetřit, proto je lepší se vkládání nedůvěryhodných dat do těchto kontextů úplně vyhnout. Patří mezi ně třeba vnořené kontexty, kdy je jeden kontext spouštěn uvnitř jiného. Příkladem vnořeného kontextu může být URL v JavaScriptovém kódu.

Další nebezpečnou praktikou je přijímání a následné spouštění JavaScriptového kódu z neznámého zdroje.

7.2 Balíčky třetích stran

V dnešní době je webové prostředí postaveno na open-source knihovnách a balíčcích. Open-source knihoven využívá obrovské množství aplikací, včetně těch největších, jakými jsou YouTube, LinkedIn nebo Microsoft Office Web. Existuje také celá řada společností, které své projekty nabízí v open-

source podobě a následně vydělávají na službách a údržbě, kterou pro tyto produkty poskytují. Příkladem takových projektů jsou například Docker a GitLab.

Závislost na open-source komponentách však představuje značné bezpečností riziko. U open-source kódu není vždy možné zjistit, jak přísnými audity kód prošel, pokud vůbec nějakými. Další komplikací jsou velmi časté změny open-source kódu, takže je každá provedená bezpečnostní analýza relevantní jen po velmi krátkou dobu a testovat kód po každé přidané změně je značně nákladné. [30]

Existuje několik způsobů, jak integrovat open-source komponenty do vlastní aplikace. Komponentu je například možné za běhu aplikace stáhnout z tzv. *content delivery network*. Další možností je potřebnou komponentu integrovat přímo do kódu projektu, tedy zkopírovat kód komponenty a vložit ho do kódu aplikace. V JavaScriptovém ekosystému je ale pro distribuci open-source komponent zdaleka nejčastěji používán registr npm. Zveřejnění vlastního balíčku je velmi snadné (viz podkapitola 3.7.1) a nepodléhá žádné bezpečností analýze. Absence bezpečnostních mechanismů je jedním z aspektů, které v minulosti způsobily několik incidentů.

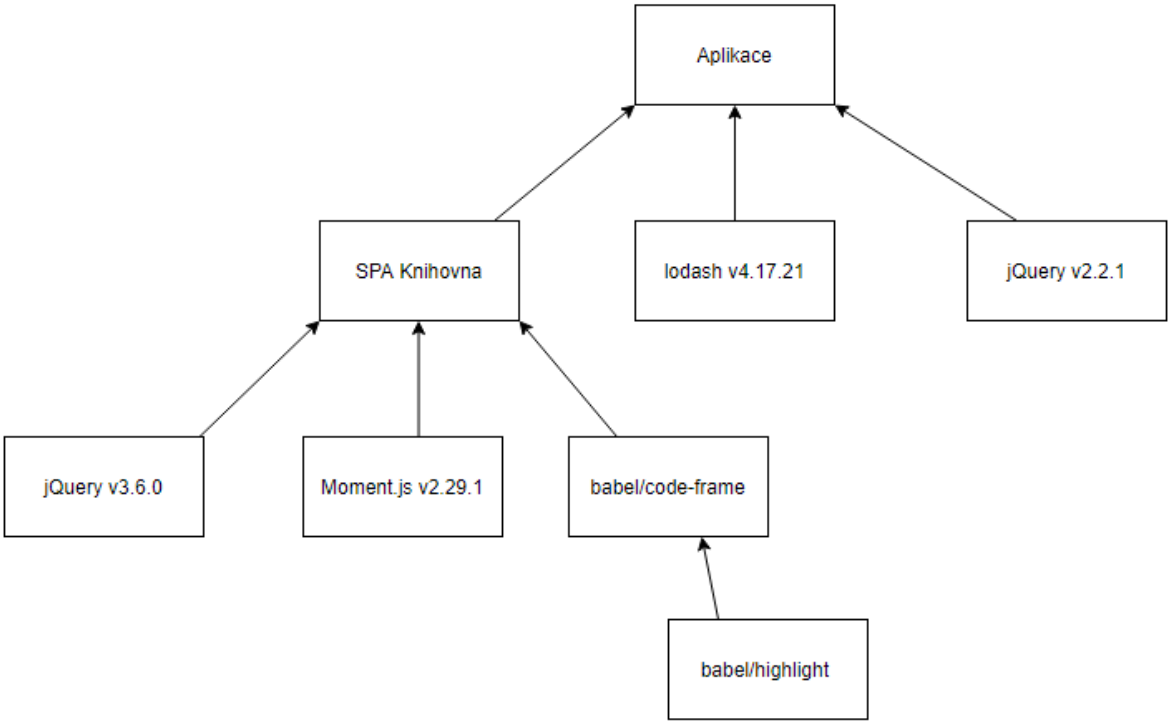
Jedním z incidentů bylo odstranění balíčku *left-pad* z registru npm v roce 2016. Tento balíček, který obsahoval jen pár desítek řádků kódu, byl spravován pouze jedním vývojářem a jakožto přímá nebo nepřímá závislost byl obsažen v milionech aplikací. Jeho odstranění tak mělo v daných aplikacích za následek výpadky v jejich CI/CD pipeline. Jako reakci na tento incident npm znemožnilo odstraňovat balíčky, u kterých od jejich publikování uběhlo více než 72 hodin. [31, 30]

V roce 2018 se odehrály další vážné incidenty. Prvním bylo odcizení účtu, pod kterým byl udržován balíček *eslint-scope*. Po odcizení účtu nahradil útočník balíček vlastním skriptem, který měl za cíl krást a odesílat citlivá data z počítačů, na kterých byl eslint-scope nainstalován. Ve stejném roce byl napaden také balíček *event-stream*, do kterého útočník přidal závislost na komponentě *flatmap-stream*. Tato komponenta obsahovala kód, který kradl bitcoinové peněženky uživatelů. [30]

7.2.1 Obrana

Balíčky třetích stran mají své vlastní závislosti. Tyto závislosti se nazývají balíčky čtvrtých stran. Balíčky čtvrtých stran mohou mít rovněž další závislosti atd. Vztahy mezi závislostmi lze znázornit tzv. stromem závislostí. Strom závislostí (viz obrázek 7.2) je z pohledu bezpečnosti možné analyzovat jak manuálně, tak automatizovaně. Manuální analýza může být v některých případech vhodnou volbou, nicméně stromy závislostí reálných aplikací jsou většinou příliš rozsáhlé, proto je potřeba analýzu automatizovat. Nejjednodušším způsobem, jak v této stromové struktuře najít zranitelnosti je iterativně ji procházet a porovnávat jednotlivé závislosti s databázemi CVE. Node package manager nabízí pro tento účel příkaz *npm audit*, existují však alternativy jako *Snyk Open Source Security Management* nebo open-source knihovna *Retire.js*. [30]

Npm nabízí ještě jeden způsob, jak závislosti kontrolovat. Při instalaci balíčků generuje soubor s názvem *package-lock.json*. Soubor obsahuje strom závislostí a zajišťuje, že se při instalaci a sestavování projektu využijí přesně definované verze závislostí. [32]



Obrázek 7.2: Příklad stromu závislostí aplikace.

7.3 Škodlivé balíčky

Společnost Snyk uvádí, že v roce 2019 byly škodlivé balíčky druhou nejčastěji hlášenou zranitelností v open-source komunitě. Jako škodlivé se označují takové balíčky, které jsou úmyslně vytvořeny tak, aby zanesly zranitelnosti do aplikace, která je použije. Může se však také jednat o známé a důvěryhodné balíčky, které byly kontaminovány zdrojovým kódem útočníka. [2]

7.3.1 Typosquatting

Vývojáři webových aplikací běžně používají nástroje pro správu balíčků (package managers). V případě JavaScriptu je nejrozšířenějším nástrojem tohoto typu dříve zmíněný node package manager. Pro stažení a instalaci balíčku se všemi jeho závislostmi stačí v příkazovém řádku použít příkaz *npm install* následovaný jménem balíčku. Typosquatting je typ útoku, při kterém se útočník snaží svůj škodlivý balíček pojmenovat tak, aby se jeho jméno co nejblíže podobalo jménu nějakého populárního

Tabulka 7.1: Ukázka reálných případů útoku typosquatting.

Původní balíček	Škodlivý balíček
axios	axioss
mongodb	mogodb
mongoose	mogoose
lodash	loadsh

balíčku. Snaží se tak o to, aby si oběť při překlepu ve psaní jména balíčku stáhla právě škodlivý balíček. Reálné příklady jsou vyobrazeny v tabulce 7.1. [33]

Jako obranu proti tomuto útoku byl vyvinut například nástroj *SpellBound*. SpellBound je spuštěn po tom, co *npm* dokončí sestavování stromu závislostí. Pro každý balíček, který má být nainstalován provede SpellBound kontrolu, ve které posuzuje popularitu balíčku a hledá v npm registru balíčky s podobným jménem. Pokud najde balíček s dostatečně podobným jménem, který je zároveň populárnější než instalovaný balíček, pak právě instalovaný balíček označí za podezřelý. Detailní popis metodiky, pomocí které SpellBound určuje popularitu a podobnost, je definován v článku [33].

7.4 Malvertising

Název malvertising vznikl spojením slov *malware* a *advertising*. Jedná se o distribuci malwaru pomocí reklam. Tento způsob šíření má z pohledu útočníka dvě výhody. První výhodou je, že uživatelé bývají méně obezřetní vůči škodlivým reklamám a odkazům, které jsou umístěné na legitimních a známých webech. Druhá výhoda spočívá v tom, že útočník nemusí získat přístup na server aplikace, stačí zaslat nebezpečnou reklamu zprostředkovateli reklamních služeb, tedy reklamní agentuře nebo serveru. Útočník pak už jen musí zaplatit stanovenou cenu za počet kliknutí. Škodlivý kód může být buď integrován přímo v reklamě samotné a spuštěn rovnou ve webové aplikaci, která reklamu zobrazuje, nebo až ve webové stránce, na kterou reklama odkazuje. Po kliknutí na reklamu je uživatel přesměrován na webovou stránku útočníka, která obsahuje skripty, které se buď uživateli v prohlížeči spustí okamžitě po načtení stránky, nebo až po provedení aktivační akce. V minulosti byly takovéto reklamy nalezeny například v populárních aplikacích jakými jsou Spotify nebo New York Times. [34]

Reklamní servery se těmto útokům snaží předcházet monitorováním a analýzou kódu. Tento proces je však velmi nákladný na čas (jednotky hodin na jeden případ) a infrastrukturu. V případě využití externích nástrojů je navíc často pro jejich legální použití nutné zaplatit licencování. Jedním ze způsobů, jak se tomuto procesu vyhnout je zakázat reklamám, aby obsahovaly skripty. Útočník se pak musí spolehnout na to, že uživatel na reklamu klikne a bude přesměrován na infikovanou stránku. Tento přístup aplikuje například společnost Facebook. Studie [35] uvádí datovou sadu 5 milionů Facebookových reklam, z nichž 0,17% bylo identifikováno jako nebezpečné. Studie zároveň uvádí,

že i přes malé procento nebezpečných reklam bylo těmto reklamám vystaveno až 33% uživatelů z jejich datové sady. [35, 34]

Běžný uživatel internetu má kromě základní kybernetické hygieny několik možností, jak se malvertisingu bránit. Moderní prohlížeče nabízí možnost tzv. *click-to-play* pluginů, které zabraňují běhu Flashe a Javy bez potvrzení uživatele. Společnost Malwarebytes také doporučuje nainstalovat rozšíření, které reklamy blokuje (anglicky ad blockers). Tato rozšíření však mohou být na některých doménách zakázána, protože zamezují doménám využívat příjmů z reklam. [36]

7.5 Phishing

Phishing je typ útoku spadajícího do kategorie sociálního inženýrství, při kterém se útočník vydává za důvěryhodnou instituci či osobu ve snaze získat citlivá data oběti. Phishingových útoků je celá řada, ale obecně je můžeme rozdělit do dvou kategorií:

7.5.1 Spam phishing

Útočník neútočí na konkrétní osobu, naopak se snaží zasáhnout co nejširší spektrum obětí. Útočníci se snaží zapůsobit na emoce, proto se v roce 2020 jedním z oblíbených témat phishingových kampaní stal Covid-19. Podvodné emaily například nabízely slevy na daních pro podnikatelé postižené vládními restrikcemi, varovaly před nárůstem covid-pozitivních případů v dané lokalitě, nebo nabízely testování. Zaměstnanci ve firmách dostávali falešné emaily obsahující pokyny pro chování na pracovišti v době pandemie apod. Článek [37] uvádí, že jedním z trendů v oblasti malware se v roce 2020 stal ransomware, který ve stejném roce způsobil škodu bezmála 20 miliard dolarů. Článek dále zmiňuje, že 26% ransomwarových útoků bylo doručeno právě prostřednictvím phishingových emailů. [37]

7.5.2 Spear phishing

Útok je cílen na určitou organizaci nebo osobu, často s vysokým postavením, kde je větší potenciál pro zisk. Existuje mnoho případů, kde byli útočníci s touto technikou úspěšní. Mezi nejznámější případy patří phishingová kampaň, při které v roce 2013 útočníci zacílili na Facebook a Google. Obě společnosti využívaly služeb firmy Quanta, útočníci se proto za tuto firmu vydávali a jejím jménem rozesílali zaměstnancům Googlu a Facebooku faktury, jejichž zaplacením přišly společnosti dohromady o více než 100 milonů dolarů. Spear phishing se v poslední době stává nejoblíbenější variantou phishingu, protože je snadnější detekovat masové spamy s generickými zprávami, než sofistikované zprávy cílené na jednotky, maximálně desítky obětí. [38, 37]

Phishing je jedním z nejefektivnějších a zároveň technicky nejjednodušších útoků na webu. Útočník se nesnaží najít technickou zranitelnost informačního systému, v jeho zájmu je využít lidské chyby a emocí.

Jedním z nejběžnějších prostředí pro phishing je elektronická pošta. Útočník imituje osobu nebo organizaci, které oběť důvěřuje a jejím jménem zašle email obsahující zprávu, která oběť vyděsí a donutí na ni zareagovat. Zpráva často obsahuje odkaz na webovou stránku a pokyny, podle kterých se má oběť na dané stránce řídit.

Základní obranou proti emailovým phishingovým útokům je před stažením souboru nebo kliknutím na odkaz prověřit emailovou adresu odesílatele a neotevírat nic, kde chybí jistota legitimního zdroje. Dalším krokem je kontrola samotného odkazu, zda je stránka zabezpečená protokolem HTTPS a má platný certifikát. Pokud je email, nebo odkaz podezřelý, je na místě do prohlížeče zkopírovat název nebo část obsahu emailu a zjistit, jestli je email součástí phishingové kampaně. Existují emailové filtry, které se snaží filtrovat phishingové a spam emaily. Pro většinu prohlížečů jsou navíc dostupná rozšíření, která dokáží zkontrolovat, zda je odkaz bezpečný. [39]

Kapitola 8

Teorie útoků a zranitelností

Předmětem kapitoly je představit čtenáři vybrané útoky a zranitelnosti, na které práce dále navazuje v kapitole 9.

8.1 Prototype pollution

Společnost Snyk uvádí, že v roce 2019 bylo zaznamenáno méně než 25 případů knihoven obsahujících zranitelnost tohoto typu. Některé z těchto knihoven, jako jsou jQuery nebo Lodash jsou však extrémně populární, proto tato zranitelnost zasáhla téměř 27% projektů, které společnost Snyk v témže roce skenovala. Výpis 8.1 demonstruje tuto zranitelnost v balíčku Lodash. [2]

```
const mergeFn = require('lodash').defaultsDeep;
const payload = '{"constructor": {"prototype": {"a0": true}}}'

function check() {
    mergeFn({}, JSON.parse(payload));
    if (({})[`a0`] === true) {
        console.log(`Vulnerable to Prototype Pollution via ${payload}`);
    }
  }

check();
```

Listing 8.1: Ukázka zranitelnosti prototype pollution v populárním balíčku Lodash (CVE-2019-10744). [40]

V kapitole 3.6 bylo stručně popsáno, jak v fungují prototypy v jazyce JavaScript. Zranitelnost zvaná prototype pollution je založená na zneužití právě tohoto mechanismu. Pokud zdrojový kód

knihovny dovoluje útočníkovi přepsat prototypy JavaScriptových objektů, pak se aplikace využívající danou knihovnu vystavuje různým typům útoků.

Jednou z možností je *Denial of Service*. JavaScriptové prototypy obsahují některé základní metody, jako například metodu *toString()*, která vrací řetězec znaků reprezentující daný objekt. Pokud útočník tuto metodu v prototypu objektu nahradí nečekanou hodnotou, pak se v případě volání metody vyskytne v aplikaci chyba, která může aplikaci zpomalovat nebo dokonce znepřístupnit.

V jazyce JavaScript existují metody, které jako svůj parametr přijímají řetězec znaků a ten následně při svém vykonávání spustí, jako by to byl JavaScriptový kód. Pokud knihovna útočníkem pozměněný prototyp dosadí jako parametr jedné z těchto metod (např. *eval()* nebo *setInterval()*), může dojít k vykonání útočníkova kódu v kontextu aplikace.

Třetím typem útoku je *property injection*, kdy útočník nahradí vlastnost, která má pro aplikaci informační hodnotu. Příkladem může být vlastnost, která určuje zdali má daný uživatel administrátorská práva. [41]

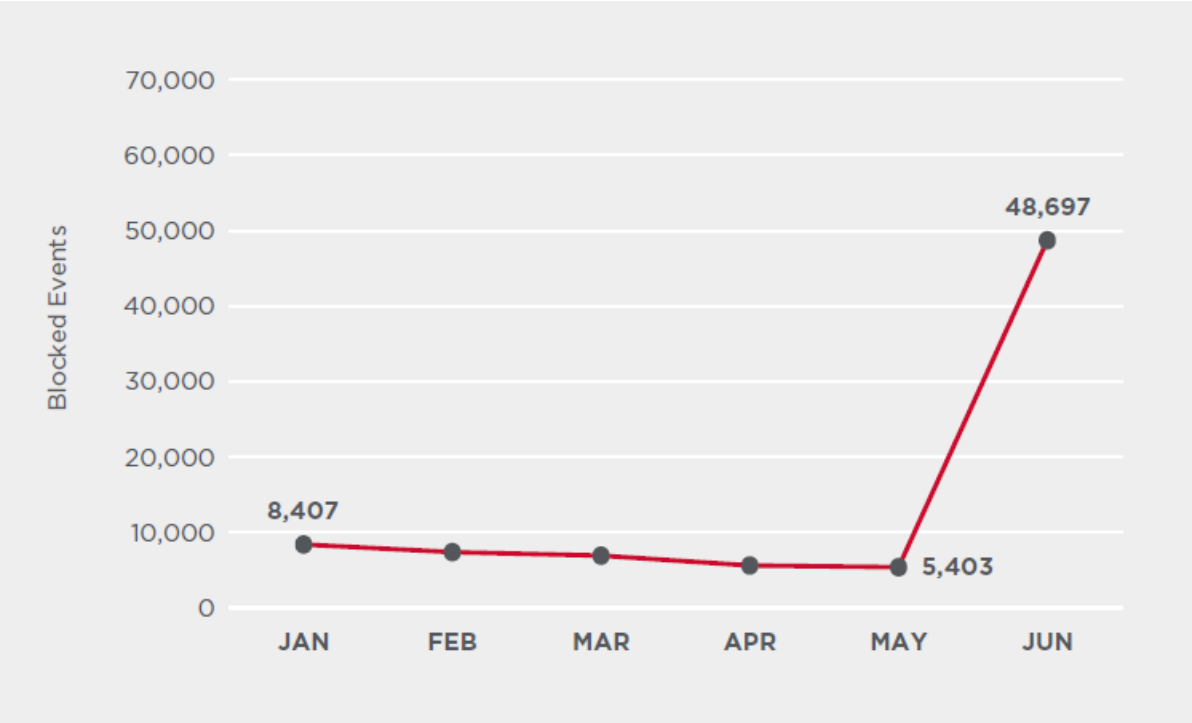
8.2 Cryptojacking

Název tohoto útoku vznikl spojením slov *cryptocurrency* a *hijacking*, tedy odcizení kryptoměn. Principem útoku je využití výpočetních prostředků oběti k těžbě kryptoměn bez jejího souhlasu. Cryptojacking má nejčastěji za následek například zpomalování napadených přístrojů, rychlejší vybíjení baterií nebo přehřívání. Tento typ útoku můžeme dále rozdělit do dvou kategorií. Názvem *file-based* je těžba kryptoměn označována tehdy, když je k ní využit spustitelný soubor na daném počítači. Oproti tomu *browser-based* je těžba odehrávající se ve webovém prohlížeči za pomoci skriptovacích jazyků. [42]

Oblíbenost tohoto útoku roste a klesá především v závislosti na cenách daných kryptoměn a na velikosti výpočetního výkonu potřebného k jejich těžbě. V roce 2017 zažila těžba v prohlížeči nebývalý růst. Byla spuštěna služba *Coinhive*, která nabízela předpřipravené skripty pro těžbu kryptoměn v prohlížeči jakožto alternativu k internetovým reklamám. Skripty však brzy začali zneužívat útočníci, kteří je vkládali do napadených webových aplikací. V době spuštění služby navíc rostly také ceny některých kryptoměn, například kryptoměny Monero. Se zánikem služby Coinhive v roce 2019 výrazně klesl i počet útoků v oblasti prohlížečů. Problém nedobrovolné těžby kryptoměn však nezmizel úplně. Důkazem je například nárůst útoků zablokovaných společností Symantec v roce 2020, který je vyobrazen na obrázku 8.1. [42, 43, 44]

8.3 Obfuskace

Obfuskace zdrojového kódu je technika, jejíž cílem je učinit zdrojový kód pro člověka nečitelným, ale zároveň zachovat jeho funkcionalitu. Takový kód je pak velmi obtížně analyzovat nebo ho podrobit



Obrázek 8.1: Nárůst cryptojacking útoků v prohlížečích detekovaný v roce 2020 společností Symantec

reverznímu inženýrství. Obfuskace je legitimní způsob, jak ukrýt a chránit logiku aplikace, tento postup je nicméně možné aplikovat také v případě škodlivého kódu. [45]

V jazyce JavaScript je několik způsobů, jak zdrojový kód obfuskovat tak, aby unikl detekci. Tato podkapitola je věnována popisu právě těchto technik.

8.3.1 Zakódování příkazů

Jedna z možností, jak učinit skript pro lidské oko nečitelným, je nějakým způsobem ho zakódovat. Prvním ze způsobů je převést JavaScriptový kód do hexadecimální podoby. Pro dekódování takového skriptu existuje několik JavaScriptových metod, jakými jsou například *unescape()* nebo *decodeURI()*. Obě metody se běžně využívají pro dekódování URI a URL, které převádí nebezpečné ASCII znaky na znak % následovaný dvěma hexadecimálními čísly. Dekódovaný řetězec znaků je následně možné spustit jakožto parametr metody *eval()*. Tento způsob je vyobrazen ve výpisu 8.2.

```
//Vypíše zprávu "Test"
eval("alert('Test')");

//Dekóduje hexadecimální řetězec a následně vypíše zprávu "Test"
eval(decodeURI("%61%6c%65%72%74%28%27%74%65%73%74%27%29"));
```

Listing 8.2: Ukázka ekvivalentních příkazů v čitelné a hexadecimální formě.

Další možností je použít kódování base64, viz výpis 8.3. Base64 se používá pro kódování binárních dat tam, kde lze pracovat pouze s ASCII znaky. Příkladem může být vkládání obsahu obrázku do HTML stránky. V JavaScriptu existují pro kódování a dekódování base64 metody *btoa()* a *atob()*. [46]

```
//Vypíše zprávu "Test"
eval("alert('Test')");

//Dekóduje base64 řetězec a následně vypíše zprávu "Test"
eval(atob("YWxlcnQoJ1Rlc3QnKQ=="));
```

Listing 8.3: Ukázka ekvivalentních příkazů v čitelné a base64 formě.

8.3.2 Manipulace s řetězci

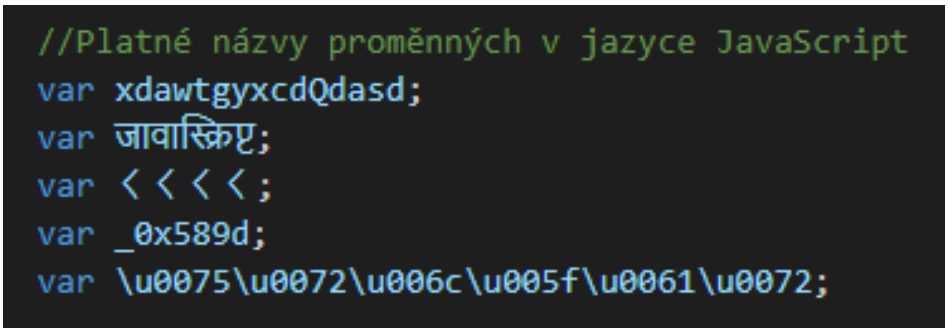
Pro obfuskaci lze využít například zdánlivě nesmyslný řetězec, deklarovaný na začátku skriptu. Ten je pak pomocí různých metod a operací přetvořen na řetězec reprezentující příkaz nebo jméno vlastnosti objektu. JavaScript má nativně hned několik takových metod. Příkladem mohou být metody *split()*, *replace()*, *concat()* nebo *substring()*. Podobný postup může být aplikován na pole řetězců, kdy je obsah daného pole v cyklu sčítán, prohazován apod. Výpis 8.4 ilustruje triviální manipulaci s polem řetězců. [46]

```
let xaDsdX = ["loXdcca", "option"];
let pojkDS = xaDsdX[0].substring(0,2) + xaDsdX[0].charAt(4) + xaDsdX[0].charAt(6)
    + "@" + xaDsdX[1].split("i").slice(-1)[0];
window[pojkDS.replace("@","t")] = "Webová adresa";
//window["location"] = "Webová adresa";
```

Listing 8.4: Ukázka triviální obfuskace pomocí manipulace s řetězci.

8.3.3 Obfuskace názvů metod a proměnných

Obfuskovat lze také názvy metod a proměnných. JavaScript povoluje pro identifikátory proměnných a metod také znaky ze sady unicode. Obrázek 8.2 demonstruje několik způsobů obfuskace proměnné. [47] [46]



Obrázek 8.2: Ukázka obfuskace názvů proměnných.

8.3.4 Minifikace

Minifikace je způsob přepsání kódu do kompaktnější podoby, která zabírá méně paměti a zároveň zachovává svou funkcionalitu. Součástí procesu je zkracování názvů proměnných a metod, dále pak odstranění nepotřebných znaků a komentářů. Jedná se o legitimní způsob, jak zredukovat velikost balíčku za účelem šetření síťových zdrojů. [45]

```
//Funkce před minifikací
function soucetCisel(prvniHodnota, druhaHodnota) {
    return prvniHodnota + druhaHodnota;
}
//Funkce po minifikaci
function soucetCisel(e,n){return e+n}
```

Listing 8.5: Ukázka triviální minifikace funkce.

8.4 Otisk prohlížeče

Některé webové aplikace používají skripty, které sbírají informace o prohlížeči uživatele. Jedná se například o časové pásmo, verzi prohlížeče, jazyková nastavení, operační systém a mnoho dalších. Kolekce těchto atributů se označuje jako *otisk prohlížeče* (z anglického *fingerprint*) a je možné na jeho základě uživatele identifikovat s vysokou mírou přesnosti. Na tento způsob sbírání informací je oproti použití tzv. *tracking cookies* aplikováno méně legislativních regulací. Otisk prohlížeče je navíc na rozdíl od cookies nemožné vymazat a nelze jej zcela obejít ani prohlížením v anonymním režimu nebo použitím VPN či tzv. *ad blockeru* (sofwtaru na blokování reklam). [48]

Některé prohlížeče a služby již implementují ochranné prvky proti tomuto typu sbírání dat. Služby obvykle využívají kombinaci následujících technik:

- *Generalizace* je technika, při které se hodnoty atributů upraví tak, aby byly co nejméně jedinečné (například změna jazykového nastavení na angličtinu), díky tomu je pak náročnější uživatele přesně identifikovat.
- *Randomizace* periodicky mění hodnoty atributů a s nimi se mění i daný otisk.
- *Skrytí* zamezuje webovým aplikacím přečtení konkrétních atributů.

Tor je příkladem prohlížeče, který skutečně používá kombinaci výše uvedených přístupů. Tor část atributů modifikuje a zbytek před aplikacemi skrývá. Atributy modifikuje pokaždé stejně, pro aplikaci jsou tak uživatelé Toru na základě otisku prohlížeče nerozlišitelní. Na obrázku 8.3 je zobrazeno srovnání otisku prohlížečů Firefox a Tor. [49]

Attribute	Value
User agent	Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:67.0) Gecko/20100101 Firefox/67.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Content encoding	gzip, deflate, br
Content language	en-US,en;q=0.5
List of plugins	
Platform	Linux x86_64
Cookies enabled	yes
Do Not Track	yes
Timezone	-120
Screen resolution	1920x1080x24
Use of local storage	yes
Use of session storage	yes
Canvas	Cwm fjordbank glyphs vext quiz, 😃 Cwm fjordbank glyphs vext quiz, 😃
WebGL Vendor	Intel Open Source Technology Center
WebGL Renderer	Mesa DRI Intel(R) UHD Graphics 620 (Kabylake GT2)

Attribute	Value
User agent	Mozilla/5.0 (Windows NT 6.1; rv:60.0) Gecko/20100101 Firefox/60.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Content encoding	gzip, deflate
Content language	en-US,en;q=0.5
List of plugins	
Platform	Linux x86_64
Cookies enabled	yes
Do Not Track	NC
Timezone	0
Screen resolution	1000x900x24
Use of local storage	yes
Use of session storage	yes
Canvas	
WebGL Vendor	Not supported
WebGL Renderer	Not supported

Obrázek 8.3: Porovnání otisku prohlížeče Firefox (vlevo) a prohlížeče Tor (vpravo). [49]

Kapitola 9

Experimenty

Kapitola detailně popisuje dva praktické experimenty, které byly v rámci práce provedeny. První z nich se zabývá testováním existujícího vzorku malware, jehož obdoba se v roce 2016 šířila v aplikaci Facebook Messenger. Druhý experiment je zaměřen na implementaci vlastních škodlivých skriptů a následné testování detekce těchto skriptů pomocí nástrojů z kapitoly 6.

9.1 Experiment 1: Testování existujícího vzorku malware

V roce 2016 se na platformě Facebook odehrála spamová kampaň, která šířila malware *Nemucod*. Nemucod je tzv. *downloader*, který útočníci používají pro stažení různých dalších typů malware. Předmětem experimentu však není samotný Nemucod, ale JavaScriptový kód, který se podílel na jeho šíření. Tento silně obfuskovaný skript byl uživatelům rozesílán ve formě SVG (Scalable Vector Graphics) souboru. SVG soubory definují vektorovou grafiku, lze do nich však také vložit JavaScriptový kód. Po otevření souboru skript uživatele přesměroval na webovou stránku útočníka, která kopírovala vzhled služby YouTube. Stránka se následně pokoušela uživatelům prohlížeče Chrome nainstalovat rozšíření s názvem *One*, které už stáhlo samotný Nemucod.

Samotný skript byl šířen v několika verzích, ale jeho princip zůstává stále stejný. Na začátku skriptu je hlavička identifikující SVG soubor (viz obrázek 9.1). Skript pak začíná definováním několika znakových řetězců (viz obrázek 9.2), které jsou následně pomocí operací s datovou strukturou pole iterativně přetvořeny na parametry objektu *window* a adresu útočníkovy stránky (viz obrázek 9.3).

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <circle cx="250" cy="250" r="50" fill="red" />
```

Listing 9.1: Hlavička SVG souboru.

```
function egvvdi(rrdpc,qutqzy,thhxs){
     var pcpuku = "tkEm.gR7BAZVf:eX?GJKsidj0/=
         03h4HNS69_oDYUMpI1PrFnv5zax8lCbuc2TLy";
     var gpepc = ["vEN=Bat32mlyCSczpR_I6Hrof7:u\/?gsJZeP9bixAMU1DGdKO0YLVFhX48j.
         kTn5","6\/=yps2T.fRo7amrFL9CXIZOAUYu0K3ed1Sgnkvxb4PjiG:JzBHDVMt5l_8NhEc?"
         ,"Zpl6rNDfa8nXoHU5IFmjui9JybMCEh1eP47:S\/0V2RBk=YLtcKGTz?3sx.vOgA_d","Sf=
         T:DOl\/v5tM9UPsyRJ1GLz.d3VpYrk6o07gHajXxNubI8mK_2?EnFc4ZAhiBCe","
         Ziok5c_IXFdl6TuYHrvRmNVsP1jDth9y3.0KpznSA=fgaG?7JLx84bUeMC\/B2O:E"];
     var gglidb = "";
     var wsduw = 0;
     while(gpepc[wsduw]){
       wsduw++;

   ...
}
```

Listing 9.2: Začátek obfuskovaného skriptu v SVG souboru.

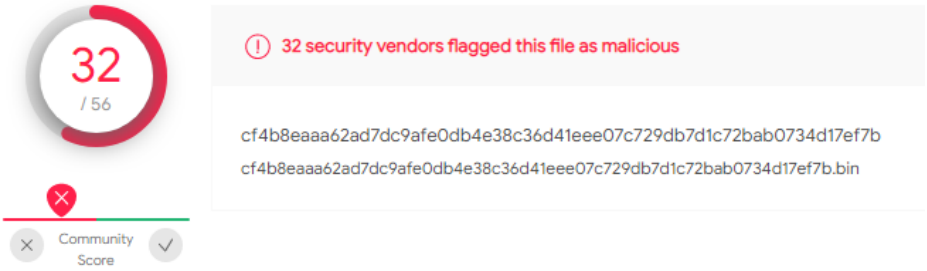
```
//Obfuskovaná verze
var vhrtt = window;
vhrtt[xvwdas][tflryl][ehudls] = egvvdi("c7/As.mp1?EPUYZvPHd1=kv8NrpOHk7PhN1v:
    v1ZBPhN",13,false);

//Deobfuskovaný ekvivalent
window[top][location][href] = "URL adresa útočníkova webu";
```

Listing 9.3: Modifikace window objektu.

9.1.1 Cíl experimentu

Služba Virus Total ukazuje, že 32 z 59 antivirových produktů na této platformě dokáže skript detekovat (viz obrázek 9.1). Cílem experimentu je zjistit, jak moc je potřeba daný skript modifikovat tak, aby se vyvaroval detekci pokud možno u všech antivirových produktů na Virus Total. Zároveň musí být zachována jeho funkcionalita. S ohledem na metody v kapitole 5 se dá předpokládat, že skript bude nutné modifikovat alespoň na několika místech, aby se předešlo detekci u všech antivirových služeb. Zároveň bude pravděpodobně nutné pozměnit i jeho chování.



Obrázek 9.1: Virus Total detekce pro Nemucod downloader a jeho hash.

9.1.2 Postup

V průběhu experimentu byl skript modifikován různými způsoby a vždy byl zaznamenán výsledek. Tyto způsoby byly následně kombinovány a výsledek kombinací byl rovněž zaznamenán. Jednotlivé kroky a jejich výsledky jsou vyobrazeny v tabulce 9.1. První kroky sestávaly z drobných úprav jako změna jména proměnné, nebo přidání nové proměnné. Cílem těchto malých úprav (viz výpis 9.4) bylo zjistit, zdali některé z antivirových služeb používají pouze detekci na bázi jednoduchých signatur. Nová logika a funkce byly přidány ve formě triviálních, několikařádkových funkcí. Funkce prováděly jednoduché matematické operace a přistupovaly k tzv. *webStorage* v prohlížeči. Jejich počet byl postupně navyšován, v posledních krocích experimentu pak byla pro téměř každý řádek původního skriptu přítomna jedna z těchto funkcí. Nové chování v podobě funkcí (viz výpis 9.5) bylo přidáno kvůli antivirům, které využívají metod heuristické detekce na základě chování skriptu. Jedním z kroků byla také úprava SVG části souboru, kdy byl SVG obrazec nahrazen několika jinými obrazci. Skript byl dále minifikován s pomocí online nástroje [50]. Konečným krokem bylo spojení předchozích technik.

```
//Původní kód
var icmvp = 0;
while(httur[icmvp]){
...
  while(fjtps[cdhhs]){
    if(fjtps[cdhhs] == httur[icmvp]){
      rmbtqy = cdhhs;
      break;
    }
    cdhhs++;
  }
...
  icmvp++;
```

```
//Změna jména proměnné icmvp na nnnmnb
var nnnmnb = 0;
while(httur[nnnmnb]){
...
  while(fjtps[cdhhs]){
    if(fjtps[cdhhs] == httur[nnnmnb]){
      rmbtqy = cdhhs;
      break;
    }
    cdhhs++;
  }
...
  nnnmnb++;
```

```
//Přidání proměnné xppof
var xppof;
var icmvp = 0;
while(httur[icmvp]){
...
  while(fjtps[cdhhs]){
    if(fjtps[cdhhs] == httur[icmvp]){
        ]){
      rmbtqy = cdhhs;
      break;
    }
    cdhhs++;
  }
...
  icmvp++;
}
```

```
//Změna jmen všech proměnných

var nnnmnb = 0;
while(hjkhjkgf[nnnmnb]){
...
  while(qweraxd[bvbndg]){
    if(qweraxd[bvbndg] == hjkhjkgf[nnnmnb

      nhjegdfg = bvbndg;
      break;
    }
    bvbndg++;
   }
...
  nnnmnb++;
}
```

Listing 9.4: Ukázka použitých technik při modifikaci vzorku malware.

```
//Triviální funkce s aritmetickými operacemi
function lkjfsd() {
    var poijlkl = 5;

    while(poijlkl > 0) {
        var mnfdsfh = poijlkl;
        poijlkl = mnfdsfh;
        poijlkl--;

    }
}
```

```
//Interakce s Web Storage
var values = [];
var keys = Object
          .keys(localStorage);
keys.forEach((key) => {
    values.push(localStorage
          .getItem(key));
});
keys = Object.keys(sessionStorage);
keys.forEach((key) => {
    values.push(sessionStorage
           .getItem(key));
});
```

Listing 9.5: Ukázka použitých prvků měnících chování.

9.1.3 Vyhodnocení výsledků

Ukázalo se, že i drobné úpravy, jako přidání jednoho řádku kódu nebo změna jména proměnné, stačily na to, aby se u některých antivirů zamezilo detekci skriptu. Signifikantní pokles detekcí byl

Tabulka 9.1: Kroky modifikace skriptu a jejich vliv na odhalení službou VirusTotal.

Číslo pokusu	Typ modifikace	Počet detekcí (z 59 možných)
0	Bez modifikace	32
1	Změna jména jedné proměnné	30
2	Přidání proměnné	28
3	Změna jmen všech proměnných	24
4	Přidání a spuštění funkce	22
5	Obalení skriptu do funkce	21
6	Změna SVG části (obrázky a hlavička)	26
7	Kombinace 3,4 a 5	19
8	Přidání nových funkcí a logiky	14
9	Minifikace JavaScript obsahu	14
10	Kombinace předchozích + nový kód	0

však zaznamenán až při krocích, ve kterých byl kód značně pozměněn, například minifikací, nebo přidáním většího množství kódu, který navíc přidal ke stávající funkcionalitě skriptu nové chování. Právě nové chování se ukázalo jako klíčové, protože v krocích, kde přidaný kód chování nerozšiřoval detekce klesaly pomaleji. Jako nejúčinější se ukázalo spojení těchto technik, které nakonec vedlo k tomu, že skript nebyl detekován žádnou z antivirových služeb na platformě Virus Total. Z tohoto faktu také vyplývá, že antiviry opravdu k detekci využívají kombinaci heuristické analýzy a signatur.

9.2 Experiment 2: Implementace a testování vlastních skriptů

Teoretická část práce nastínila závažnost problematiky bezpečnosti balíčků třetích stran a jejich závislostí. V rámci experimentu bylo implementováno několik útoků a zranitelností, které byly následně testovány nástroji uvedenými v kapitole 6. Cílem experimentu je zjistit, jak složité je pro zranitelný či nebezpečný kód uniknout detekci těchto nástrojů.

9.2.1 Postup

Postup implementace a testování všech skriptů byl následovný:

1. Implementace skriptů.
2. Testování samostatných skriptu nástroji ESLint, Virus Total a Cuckoo.

3. Přidání skriptů do souboru se všemi předešlými skripty a následné testování souboru pomocí Virus Total a Cuckoo.

Skripty byly v nástrojích Virus Total a Cuckoo testovány jako součást triviální HTML stránky, protože většina z nich zahrnuje interakci s prohlížečem a strukturou DOM. Testování skriptů je vyobrazeno v tabulkách 9.2 a 9.3.

Skript 1: Metody eval() a setInterval()

Metody *eval()* a *setInterval()* byly v práci již několikrát zmíněny. Nebezpečí metod spočívá v tom, že dokáží vykonat kód, který je jim předán ve formě řetězce znaků. Článek [51] poukazuje na četnost těchto metod ve zranitelných knihovnách a balíčcích. Skript na několika místech volá eval() a setInterval(). Volání eval() dokáže ESLint ve výchozím nastavení zachytit. Stejně tak dokáže zachytit setInterval(), pokud je této funkci předáván místo funkce jako parametr řetězec znaků. Výpis 9.6 ukazuje, jak obejít pravidlo pro funkci setInterval().

```
const TEST_STRING = "alert('Implied eval. Consider passing a function instead
    of a string.')";

//Zachyceno ESLintem - Pravidlo: (no-implied-eval) Implied eval. Consider
    passing a function instead of a string.
setInterval(TEST_STRING, INTERVAL_IN_MS);

//Projde kontrolou ESLintu
setInterval(function executeImmediately() {return TEST_STRING}(),
    INTERVAL_IN_MS);
```

Listing 9.6: Volání funkce setInterval.

Skript 2: Bitcoin miner a iframe

Zde byl skript implementován tak, aby na stránce, kde je spuštěn, vygeneroval neviditelný iframe a v něm otevřel online implementaci knihovny *Bcoin*. Bcoin je JavaScriptová knihovna pro těžbu kryptoměny Bitcoin. [52]

Skript 3: Drive-by-download

Drive by download je nechtěné stažení souboru uživatelem při návštěvě webové stránky nebo využívání internetové služby. Tato implementace vytvoří při jejím spuštění na stránce odkaz a okamžitě na něj klikne. Kliknutí stáhne HTML soubor, rovněž vygenerovaný skriptem, který obsahuje další skript. Skript ve staženém souboru po spuštění přesměruje uživatele na URL adresu obsaženou v jedné z verzí skriptu v kapitole 9.1. Obsah staženého souboru je ve skriptu tvořen pomocí JavaScriptových funkcí pro manipulaci s řetězci (viz ukázka 9.7), které pracují s řetězci uloženými v

proměnných skriptu. Přidání funkcí pro manipulaci s řetězci bylo inspirováno článkem [51], který zmiňuje jejich časté užití v nebezpečných skriptech.

```
let HTML_STRING = '<!DOCTYPE html> <html lang="en"> placeholder';
let HTML_HEAD = '<head> <meta charset="utf-8" /> </head>';
let HTML_BODY = '<body>placeholder2</body></html>';
let JS_SCRIPT = '<script> window.location = "Mailicous URL";</script>';

HTML_STRING = HTML_STRING.replace('placeholder', HTML_HEAD);
HTML_STRING = HTML_STRING.concat(HTML_BODY);
let index = HTML_STRING.indexOf('placeholder2');
let substring = HTML_STRING.substring(index, index + 'placeholder2'.length);
HTML_STRING = HTML_STRING.replace(substring, JS_SCRIPT);
```

Listing 9.7: Manipulace se stringy pomocí vestavěných JavaScriptových funkcí.

Skript 4: Keylogger

Keylogger je typ spywaru, který zaznamenává stisky kláves. Tímto způsobem je možné získat uživatelská hesla, PIN kódy, čísla kreditních karet a jakékoliv další informace, které uživatel pomocí klávesnice napíše. [53]

Tato implementace keyloggeru v prohlížeči přidává do stránky event listener, který čeká na tzv. *keydown event*. Event má vlastnost *key*, která vrací hodnotu stisknuté klávesy (včetně znaků modifikovaných například klávesou Shift). Implementace má pouze čtyři řádky (bez odesílání dat na server) a je zobrazena ve výpisu 9.8. [54]

```
let keys = '';

document.addEventListener('keydown', (event) => {
    keys += event.key;
})
```

Listing 9.8: JavaScriptová implementace keyloggeru.

Skript 5: Prototype pollution

Zranitelnost prototype pollution byla blíže popsána v kapitole 8.1. Implementace zranitelnosti nahrazuje metodu *Object.prototype.toString()* (viz výpis 9.9). Pokaždé, když je vytvořen nový objekt, přenese se na něj pomocí prototype řetězce také tato konkrétní implementace toString().

```
//Zachyceno ESLintem - Pravidlo: (no-extend-native) Object prototype is read only,
    properties should not be added.
```

```
Object.prototype.toString = () => {
       //Tato metoda se zavolá pokaždé, když objekt dědící prototype řetězec z
           Object zavolá metodu toString
}

//Ekvivalentní zápis - Projde kontrolou ESLintu
let pollutedObject = Object.prototype;
pollutedObject.toString = () => {
    //Tato metoda se zavolá pokaždé, když objekt dědící prototype řetězec z Object
        zavolá metodu toString
}
```

Listing 9.9: Implementace zranitelnosti prototype pollution.

Skript 6: Krádež Web Storage dat

Web Storage umožňuje prohlížečům ukládat data ve formě: klíč/hodnota. Web storage se dělí na dvě části:

- *Session Storage* ukládá data pro konkrétní origin (kombinace domény, protokolu a portu) po dobu běhu prohlížeče nebo záložky pro danou aplikaci. Data zůstávají uložena i po refreshi stránky, zároveň nelze uložit více než 5MB dat.

- *Local storage* je obdobou úložiště web storage, s tím rozdílem, že data zůstávají uložena i po ukončení procesu prohlížeče. Vymazat je lze pomocí JavaScriptu nebo smazáním cache v prohlížeči.

Obě úložiště se používají především pro ukládání jazykových, grafických a dalších uživatelských preferencí. Veškerá data uložená ve web storage jsou přímo dostupná z JavaScriptového kódu, proto není tento typ persistence vhodný pro ukládání citlivých dat. V okamžiku, kdy se v aplikaci vyskytne například XSS zranitelnost, jsou veškerá data dostupná útočníkovi. [55]

Šestý skript experimentu prochází jak local storage, tak session storage a ukládá všechny nalezené klíče a jejich hodnoty. Pro tento účel byla použita metoda *Object.keys()*, která jako parametr přijímá objekt a vrací pole se jmény všech vlastností objektu. Protože jsou obě úložiště v prohlížeči reprezentovány vlastními objekty, stačí metodě Object.keys() objekty předat a následně iterativně projít pomocí WebStorage API všechny nalezené klíče.

Skript 7: Otisk prohlížeče

Účelem skriptu je sbírat data o prohlížeči uživatele (viz kapitola 8.4). Pro tento účel byla zvolena open-source knihovna *fingerprintjs*. Obrázek 9.4 zobrazuje příklad kolekce atributů získaných knihovnou fingerprintjs. [56]

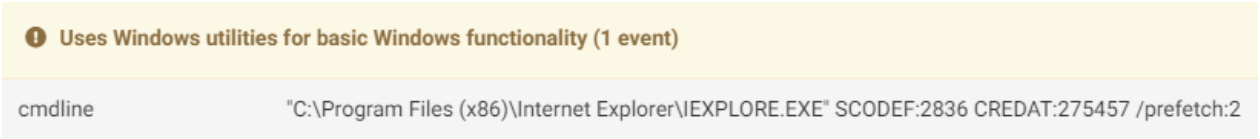
Implementace skriptu nejdříve pomocí JavaScriptové *Fetch API* načte kód knihovny do paměti jako řetězec znaků. Poté řetězec připojí k předem připravenému skriptu, který již volá jednotlivé metody knihovny fingerprintjs. Výslední skript v podobě řetězce je následně předán metodě eval(), která jej vykoná.

9.2.2 Vyhodnocení výsledků

Nástroj ESLint v základním nastavení správně zachytil zranitelnosti ve formě metod eval() a setInterval(), které mohou být využity pro vzdálené vykonání kódu. Zároveň zachytil nesprávnou manipulaci s objektovými prototypy. Tato pravidla však lze obejít. Nástroj je určen především pro statickou analýzu správnosti kódu, a tak není překvapující, že u ostatních skriptů nedokázal detekovat podezřelé chování.

Z tabulky 9.2 je patrné, že všechny skripty úspěšně prošly kontrolou služby Virus Total. Jedním z důvodů může být to, že většina skriptů v experimentu obsahuje chování, které se běžně používá v legitimních aplikacích. Logování kláves je například součástí implementace online her, interakce s WebStorage API je běžnou technikou pro ukládání uživatelských preferencí apod. Zároveň může být složité rozlišit legitimní HTTP požadavky na server od těch škodlivých.

Tabulka 9.3 na první pohled ukazuje, že nástroj Cuckoo u skriptů odhalil známky podezřelého chování. Většina skriptů byla ohodnocena známkou 1,4 a poznámkou: *This file shows some signs of potential malicious behavior*. Po bližším prozkoumání nahlášeného chování se však ukázalo, že výpis obsahuje na několika místech zavádějící informace. U části skriptů identifikoval Cuckoo v tzv. *process memory dump* řadu URL odkazů, které ve skriptech žádným způsobem nebyly zahrnuty. V případě keyloggeru byl dokonce skript nahlášen za *port scanning* i přes to, že žádnou takovou funkcionalitou nedisponuje. Pro všechny testované skripty byly zároveň nahlášeny stejné, obecné problémy jako spouštění prohlížeče (viz obrázek 9.2) nebo manipulace s pamětí (viz obrázek 9.3). Tyto problémy detekuje Cuckoo nezávisle na konkrétním obsahu skriptu a byly by tak detekovány i u naprosto bezpečného JavaScriptového kódu.



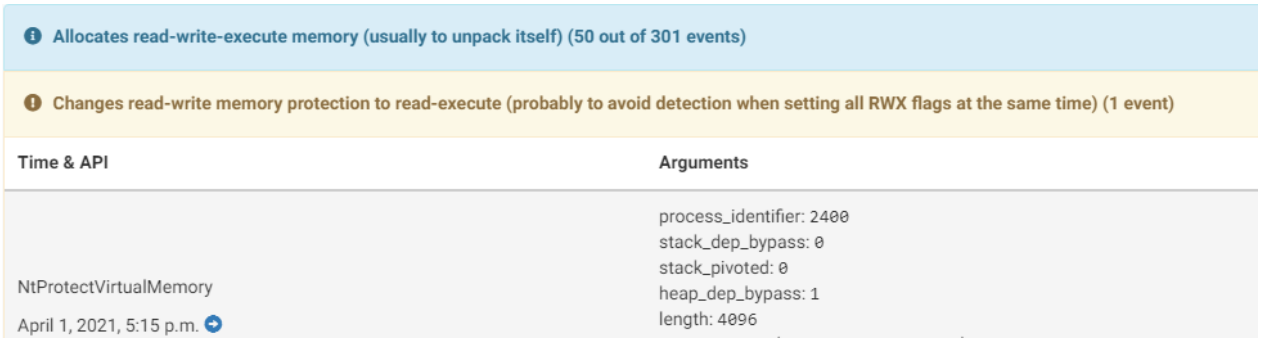
Obrázek 9.2: Problém detekovaný nástrojem Cuckoo u všech testovaných skriptů.

Tabulka 9.2: Implementace vlastních skriptů a jejich detekce službou Virus Total.

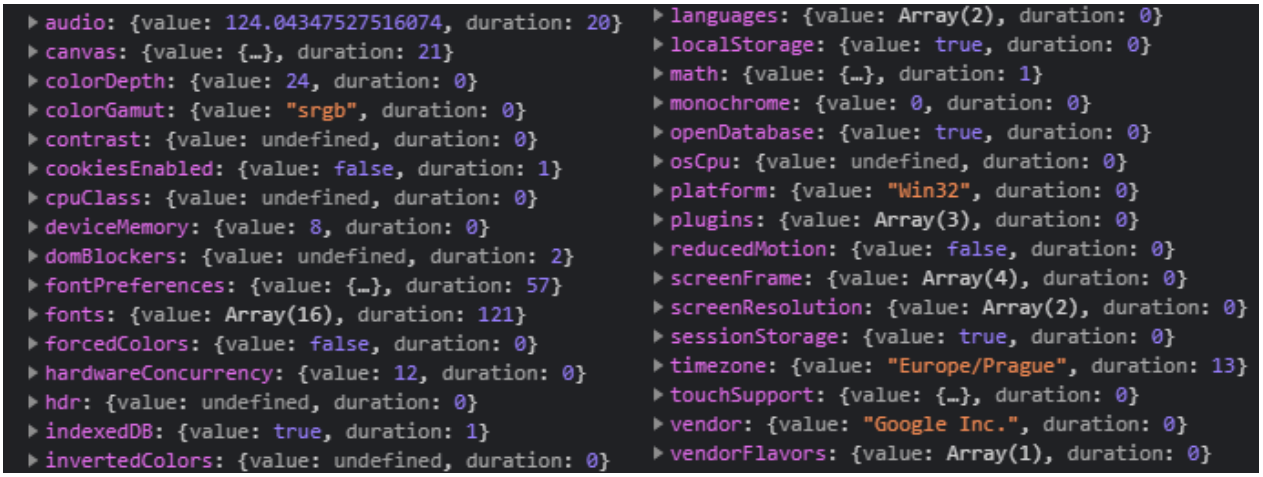
Číslo skriptu	Typ útoku/zranitelnosti	Počet detekcí (z 59 možných)
1	eval/setInterval	0
2	Bitcoin miner/iframe	0
3	Drive-by-download	0
4	Keylogger	0
5	Prototype pollution	0
6	Krádež WebStorage	0
7	Otisk prohlížeče	0
8	Kombinace předešlých	0

Tabulka 9.3: Implementace skriptů a jejich detekce službou Cuckoo.

Číslo skriptu	Typ útoku/zranitelnosti	Podezřelé chování	Skóre z 10
1	eval/setInterval	5	1,4
2	Bitcoin miner/iframe	5	1,4
3	Drive-by-download	5	1,4
4	Keylogger	6	1,6
5	Prototype pollution	5	1,4
0	Krádež WebStorage	5	1,4
7	Otisk prohlížeče	5	1,4
8	Kombinace předešlých	5	1,4



Obrázek 9.3: Problém s manipulací paměti detekovaný nástrojem Cuckoo u všech testovaných skriptů.



Obrázek 9.4: Data získaná knihovnou fingerprintjs.

Kapitola 10

Závěr

Práce byla zaměřena na popis aktuálních problémů JavaScriptového ekosystému především z hlediska bezpečnosti JavaScriptových knihoven a balíčků třetích stran. Zároveň rozebrala současná řešení daných problémů a jejich nedostatky.

První část textu uvedla čtenáře do problematiky a nastínila její rozsáhlost a závažnost. Jako příklad byl uveden balíčkový registr *npm*, ze kterého jsou denně stahovány stovky milionů balíčků. Dále byla zmíněna vysoká četnost zranitelností, když společnost Snyk objevila zranitelnosti v 77% webových stránek z jimi spravovaného vzorku čítajícího přes 400 000 webů. Jednou z nejrozšířenějších zranitelností byla zranitelnost typu prototype pollution, která byla nalezena v extrémně populárním balíčku *lodash* a zasáhla tak stovky tisíc projektů. Zmíněny byly rovněž škodlivé balíčky, které autoři vytvořili s úmyslem zanést zranitelnosti do jinak legitimních balíčků a knihoven. Následujících několik kapitol popisovalo způsoby analýzy kódu a překážky, které do tohoto procesu přináší specifické chování jazyka JavaScript. Dále byly vylíčeny metody, kterými se může nebezpečný kód šířit mezi uživatele a aplikace. V neposlední řadě pak byly vysvětleny vybrané zranitelnosti a techniky, které útočníci využívají k napadení zejména webových aplikací.

Závěrečná, praktická část práce byla věnována experimentům, které měly za cíl demonstrovat problémy současných metod analýzy JavaScriptového kódu. První experiment se zabýval testováním existujícího vzorku malware, který v roce 2016 koloval mezi uživateli v aplikaci Facebook Messenger. Stěžejním bodem experimentu bylo nalézt způsoby, jak skript skrýt před antivirovými službami. Vzorek byl pomocí několika technik úspěšně modifikován tak, že ho žádná z antivirových služeb platformy Virus Total nebyla schopná detekovat jako nebezpečný.

Druhý experiment obsahoval vlastní implementace sedmi vybraných útoků a zranitelností z předchozí části práce, které byly následně analyzovány volně dostupnými nástroji ESLint, Virus Total a Cuckoo. Funkcionalita testovaných skriptů zahrnovala například zachycení otisku prohlížeče, logování stisknutých kláves nebo neviditelnou těžbu kryptoměn v pozadí aplikace. V případě služby Virus Total nedošlo k detekci žádného ze skriptů. Cuckoo dokázalo detekovat podezřelé chování, které se však dá považovat za příliš obecné a v mnoha případech jako falešně pozitivní. Oba expe-

rimenty ukázaly potenciální nedostatky v aktuální metodice detekce škodlivého JavaScript kódu, zvlášť vzhledem k četnosti zranitelností a dosahu JavaScriptových komponent.

Prvním z problémů je možnost publikovat balíček bez jakékoli kontroly. Bylo by tedy vhodné do procesu publikace zařadit nějakou formu analýzy kódu. S ohledem na vysoký počet balíčků a frekvenci jejich aktualizací může být tento proces velmi nákladný na zdroje. Potenciálním řešením je analýzu zařadit pouze pro nejrozšířenější balíčky, případně vytvořit systém, který přenechá audit kódu na straně vývojářů balíčku.

Samotný proces analýzy JavaScriptového kódu také není bezchybný. Z experimentů plyne, že se některé služby spoléhají především na statickou analýzu kódu, v případě antivirových služeb pak na detekci na bázi signatur. Je obtížné automatizovanými způsoby rozlišit chování nebezpečného a legitimního JavaScriptového kódu. V rámci této problematiky se jako zajímavé jeví tzv. *expert systems*, které imitují rozhodovací proces lidského analytika a pokouší se klasifikovat kód tak, jak by to udělal člověk při manuální kontrole.

Technologicky nejméně náročným, zato však velmi důležitým krokem v procesu zvyšování bezpečnosti JavaScriptového ekosystému je neustálé vzdělávání vývojářů a uživatelů.

Literatura

1. *The State of Open Source Security 2017* [online] [cit. 2021-01-16]. Dostupné z: https://snyk.io/blog/launching-state-of-oss-security/.
2. *The State of Open Source Security 2020* [online] [cit. 2021-01-16]. Dostupné z: https://snyk.io/open-source-security/.
3. *Trustwave Global Security Report* [online] [cit. 2021-01-16]. Dostupné z: https://www.trustwave.com/en-us/resources/library/documents/2018-trustwave-global-security-report/.
4. *Source Code Analysis Tools. OWASP* [online] [cit. 2021-01-16]. Dostupné z: https://owasp.org/www-community/Source_Code_Analysis_Tools.
5. 2020 Index IEEE Transactions on Software Engineering Vol. 46. *IEEE Transactions on Software Engineering*. 2021, roč. 47, č. 1, s. 1–10. Dostupné z DOI: 10.1109/TSE.2020.3045901.
6. ANDREASEN, Esben; GONG, Liang; MØLLER, Anders; PRADEL, Michael; SELAKOVIC, Marija; SEN, Koushik; STAICU, Cristian-Alexandru. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 2017-09, roč. 50, č. 5. ISSN 0360-0300. Dostupné z DOI: 10.1145/3106739.
7. ZIMMERMANN, Markus; STAICU, Cristian-Alexandru; TENNY, Cam; PRADEL, Michael. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019-08, s. 995–1010. ISBN 978-1-939133-06-9. Dostupné také z: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman.
8. *The state of JavaScript frameworks security report 2019* [online] [cit. 2021-01-16]. Dostupné z: https://snyk.io/blog/javascript-frameworks-security-report-2019/.
9. *Npm-audit. Npm Docs* [online] [cit. 2021-01-16]. Dostupné z: https://docs.npmjs.com/cli/v6/commands/npm-audit.
10. FLANAGAN, David. *JavaScript : the definitive guide : master the world's most-used programming language*. Sebastopol, CA: O'Reilly Media, 2020. ISBN 978-1491952023.

11. HAVERBEKE, Marijn. *Eloquent JavaScript : a modern introduction to programming*. San Francisco: No Starch Press, 2019. ISBN 978-1593279509.

12. *About JavaScript: What is JavaScript? MDN Web Docs* [online] [cit. 2021-02-25]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript`.

13. WIRFS-BROCK, Allen; EICH, Brendan. JavaScript: The First 20 Years. *Proc. ACM Program. Lang.* 2020-06, roč. 4, č. HOPL. Dostupné z DOI: `10.1145/3386327`.

14. PAPADOPOULOS, Panagiotis; ILIA, Panagiotis; POLYCHRONAKIS, Michalis; MARKATOS, Evangelos P.; IOANNIDIS, Sotiris; VASILIADIS, Giorgos. *Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation*. 2018. Dostupné z arXiv: `1810.00464 [cs.CR]`.

15. *File and Directory Entries API. MDN Web docs* [online] [cit. 2021-02-08]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API`.

16. *Same-origin policy* [online] [cit. 2021-04-02]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`.

17. *Cross-Origin Resource Sharing (CORS)* [online] [cit. 2021-04-02]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

18. NTANTOGIAN, Christoforos; BOUNTAKAS, Panagiotis; ANTONAROPOULOS, Dimitris; PATSAKIS, Constantinos; XENAKIS, Christos. NodeXP: NOde.js server-side JavaScript injection vulnerability DEtection and eXPloitation. *Journal of Information Security and Applications*. 2021, roč. 58, s. 102752. ISSN 2214-2126. Dostupné z DOI: `https://doi.org/10.1016/j.jisa.2021.102752`.

19. SIMPSON, Kyle. *Scope and closures*. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1-4493-3558-8.

20. *About npm. Npm Docs* [online] [cit. 2021-03-03]. Dostupné z: `https://docs.npmjs.com/about-npm`.

21. *Static Code Analysis* [online] [cit. 2021-04-17]. Dostupné z: `https://owasp.org/www-community/controls/Static_Code_Analysis`.

22. SUN, Kwangwon; RYU, Sukyoung. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.* 2017-08, roč. 50, č. 4. ISSN 0360-0300. Dostupné z DOI: `10.1145/3106741`.

23. KORET, Joxean; BACHAALANY, Elias. The Antivirus Hacker's Handbook. In: 2015.

24. *About ESLint* [online] [cit. 2021-04-06]. Dostupné z: `https://eslint.org/docs/about/`.

25. *How it works* [online] [cit. 2021-04-05]. Dostupné z: `https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works`.

26. *Cuckoo* [online] [cit. 2021-04-06]. Dostupné z: https://cuckoosandbox.org/.

27. *Cross Site Scripting (XSS). OWASP* [online] [cit. 2021-01-24]. Dostupné z: https://owasp.org/www-community/attacks/xss/.

28. SHALINI, S; USHA, S. Prevention Of Cross-Site Scripting Attacks (XSS) On Web Applications In The Client Side. *Int. J. Comput. Sci. Issues*. 2011-07, roč. 8.

29. *DOM Based XSS. OWASP* [online] [cit. 2021-01-24]. Dostupné z: https://owasp.org/www-community/attacks/DOM_Based_XSS.

30. HOFFMAN, Andrew. *Web application security : exploitation and countermeasures for modern web applications / Andrew Hoffman*. O'Reilly Media, Inc, 2020. ISBN 9781492053118.

31. *Unpublishing packages from the registry* [online] [cit. 2021-04-10]. Dostupné z: https://docs.npmjs.com/unpublishing-packages-from-the-registry.

32. *Package-lock.json* [online] [cit. 2021-04-11]. Dostupné z: https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json.

33. TAYLOR, Matthew; VAIDYA, Ruturaj K.; DAVIDSON, Drew; CARLI, Lorenzo De; RASTOGI, Vaibhav. *SpellBound: Defending Against Package Typosquatting*. 2020. Dostupné z arXiv: 2003.03471 [cs.SE].

34. HUANG, C. -T.; SAKIB, M. N.; KAMHOUA, C. A.; KWIAT, K. A.; NJILLA, L. A Bayesian Game Theoretic Approach for Inspecting Web-Based Malvertising. *IEEE Transactions on Dependable and Secure Computing*. 2020, roč. 17, č. 6, s. 1257–1268. Dostupné z DOI: 10.1109/TDSC.2018.2866821.

35. ARRATE, Aritz; GONZÁLEZ-CABAÑAS, José; CUEVAS, Ángel; CUEVAS, Rubén. Malvertising in Facebook: Analysis, Quantification and Solution. *Electronics*. 2020, roč. 9, č. 8. ISSN 2079-9292. Dostupné z DOI: 10.3390/electronics9081332.

36. *Malvertising* [online] [cit. 2021-04-12]. Dostupné z: https://www.malwarebytes.com/malvertising/.

37. *Top 6 Phishing Trends of 2020* [online] [cit. 2021-04-12]. Dostupné z: https://www.vadesecure.com/en/blog/top-phishing-trends.

38. *The 5 Most Expensive Phishing Scams of all Time* [online] [cit. 2021-04-12]. Dostupné z: https://www.checkpoint.com/cyber-hub/threat-prevention/what-is-phishing/the-top-5-phishing-scams-of-all-times/.

39. *What is phishing?* [Online] [cit. 2021-04-12]. Dostupné z: https://www.malwarebytes.com/phishing/.

40. *Snyk research team discovers severe prototype pollution security vulnerabilities affecting all versions of lodash* [online] [cit. 2021-04-02]. Dostupné z: https://snyk.io/blog/snyk-research-team-discovers-severe-prototype-pollution-security-vulnerabilities-affecting-all-versions-of-lodash.

41. *Prototype Pollution: Affecting lodash package, versions <4.17.12.* [Online] [cit. 2021-02-23]. Dostupné z: https://snyk.io/vuln/SNYK-JS-LODASH-450202.

42. O'GORMAN, Brigid. *Internet Security Threat Report: Cryptojacking: A Modern Cash Cow* [online]. 2018 [cit. 2021-04-02]. Dostupné z: https://docs.broadcom.com/doc/istr-cryptojacking-modern-cash-cow-en.

43. *ENISA Threat Landscape 2020 - Cryptojacking* [online] [cit. 2021-04-02]. Dostupné z: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2020-cryptojacking.

44. *Threat Landscape Trends – Q2 2020* [online] [cit. 2021-04-02]. Dostupné z: https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/threat-landscape-trends-q2-2020.

45. SARKER, Shaown; JUECKSTOCK, Jordan; KAPRAVELOS, Alexandros. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In: *Proceedings of the ACM Internet Measurement Conference*. Virtual Event, USA: Association for Computing Machinery, 2020, s. 648–661. IMC '20. ISBN 9781450381383. Dostupné z DOI: 10.1145/3419394.3423616.

46. *CATCH ME IF YOU CAN - JAVASCRIPT OBFUSCATION* [online] [cit. 2021-04-05]. Dostupné z: https://blogs.akamai.com/sitr/2020/10/catch-me-if-you-can---javascript-obfuscation.html.

47. *Grammar and types* [online] [cit. 2021-04-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types.

48. *What Is Browser Fingerprinting and How Can You Prevent It?* [Online] [cit. 2021-04-10]. Dostupné z: https://www.avast.com/c-what-is-browser-fingerprinting.

49. *Browser Fingerprinting: An Introduction and the Challenges Ahead* [online] [cit. 2021-04-12]. Dostupné z: https://blog.torproject.org/browser-fingerprinting-introduction-and-challenges-ahead.

50. *JavaScript Minifier* [online] [cit. 2021-04-12]. Dostupné z: https://javascript-minifier.com/.

51. , wan nurul wan manan wan nurul; MOHMAD KAHAR, Mohd Nizam; ALI, Noorlin. A Survey on Current Malicious JavaScript Behavior of infected Web Content in Detection of Malicious Web pages. *IOP Conference Series: Materials Science and Engineering*. 2020-06, roč. 769, s. 012074. Dostupné z DOI: 10.1088/1757-899X/769/1/012074.

52. *Bcoin online node* [online] [cit. 2021-04-12]. Dostupné z: https://browser.c4yt.io:8080/.

53. *What is a keylogger and how do I protect myself against one?* [Online] [cit. 2021-04-12]. Dostupné z: https://us.norton.com/internetsecurity-malware-what-is-a-keylogger.html.

54. *KeyboardEvent.key* [online] [cit. 2021-04-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key.

55. *Web Storage API* [online] [cit. 2021-04-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.

56. *fingerprintjs* [online] [cit. 2021-04-12]. Dostupné z: https://github.com/fingerprintjs/fingerprintjs.

Příloha A

Příloha v IS EDISON

Skripty použité v experimentech jsou dostupné v příloze práce v následujícím formátu:

- **Experiment1** - Složka obsahující skripty testované v rámci prvního experimentu. Soubory jsou očíslovány podle čísel kroků v textu.
  - Experiment1/sample0.svg
  - Experiment1/sample1.svg
  - Experiment1/sample2.svg
  - Experiment1/sample3.svg
  - Experiment1/sample4.svg
  - Experiment1/sample5.svg
  - Experiment1/sample6.svg
  - Experiment1/sample7.svg
  - Experiment1/sample8.svg
  - Experiment1/sample9.svg
  - Experiment1/sample10.svg
- **Experiment2** - Složka obsahující skripty testované v rámci druhého experimentu. Jméno skriptu je vždy téměř totožné s názvem skriptu uvedeném v textu.
  - Experiment2/browser-fingerprinting.js
  - Experiment2/code-execution.js
  - Experiment2/drive-by-download.js
  - Experiment2/iframe.js
  - Experiment2/keylogger.js

- Experiment2/prototype-pollution.js
- Experiment2/web-storage-grabber.js